Logical Methods in Computer Science
Vol. 4 (1:3) 2008, pp. 1–39
www.lmcs-online.org

Submitted Jul. 5, 2007
Published Mar. 14, 2008

CALL-BY-VALUE TERMINATION IN THE UNTYPED λ-CALCULUS

NEIL D. JONES [a] AND NINA BOHR [b]

[a] DIKU, University of Copenhagen, Denmark
e-mail address: neil@diku.dk

[b] IT-University of Copenhagen, Denmark
e-mail address: nina@itu.dk

Abstract. A fully-automated algorithm is developed able to show that evaluation of a given untyped λ-expression will terminate under CBV (call-by-value). The "size-change principle" from first-order programs is extended to arbitrary untyped λ-expressions in two steps. The first step suffices to show CBV termination of a single, stand-alone λ-expression. The second suffices to show CBV termination of any member of a regular set of λ-expressions, defined by a tree grammar. (A simple example is a minimum function, when applied to arbitrary Church numerals.) The algorithm is sound and proven so in this paper. The Halting Problem's undecidability implies that any sound algorithm is necessarily incomplete: some λ-expressions may in fact terminate under CBV evaluation, but not be recognised as terminating.

The intensional power of the termination algorithm is reasonably high. It certifies as terminating many interesting and useful general recursive algorithms including programs with mutual recursion and parameter exchanges, and Colson's "minimum" algorithm. Further, our type-free approach allows use of the Y combinator, and so can identify as terminating a substantial subset of PCF.

Contents

1998 ACM Subject Classification: F.3.2, D.3.1.
Key words and phrases: Program analysis, Termination analysis, Untyped Lambda calculus, The Size-Change Principle.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.2168/LMCS-4 (1:3) 2008

1. Introduction

The *size-change* analysis by Lee, Jones and Ben-Amram [14] can show termination of programs whose parameter values have a well-founded size order. The method is reasonably general, easily automated, and does not require human invention of lexical or other parameter orders. It applies to first-order functional programs. This paper applies similar ideas to termination of *higher-order* programs. For simplicity and generality we focus on the simplest such language, the λ-calculus.

Contribution of this paper. Article [12] (prepared for an invited conference lecture) showed how to lift the methods of [14] to show termination of closed λ-expressions. The current paper is a journal version of [12]. It extends [12] to deal not only with a single λ-expression in isolation, but with a regular set of λ-expressions generated by a finite tree grammar. For example, we can show that a λ-expression terminates when applied to Church numerals, even though it may fail to terminate on all possible arguments. This paper includes a number of examples showing its analytical power, including programs with primitive recursion, mutual recursion and parameter exchanges, and Colson's "minimum" algorithm. Further, examples show that our type-free approach allows free use of the Y combinator, and so can identify as terminating a substantial subset of PCF.

1.1. Related work.

Jones [11] was an early paper on control-flow analysis of the untyped λ-calculus. Shivers' thesis and subsequent work [22, 23] on CFA (control flow analysis) developed this approach considerably further and applied it to the Scheme programming language. This line is closely related to the approximate semantics (static control graph) of Section 3.6 [11].

Termination of untyped programs. Papers based on [14] have used size-change graphs to find bounds on program running times (Frederiksen and Jones [5]); solved related problems, e.g., to ensure that partial evaluation will terminate (Glenstrup and Jones, Lee [10, 15]); and found more efficient (though less precise) algorithms (Lee [16]). Further, Lee's thesis [17] extends the first-order size-change method [14] to handle higher-order named combinator programs. It uses a different approach than ours, and appears to be less general.

We had anticipated from the start that our framework could naturally be extended to higher-order functional programs, e.g., functional subsets of Scheme or ML. This has since been confirmed by Sereni and Jones, first reported in [19]. Sereni's Ph.D. thesis [21] develops this direction in considerably more detail with full proofs, and also investigates problems with lazy (call-by-name) languages. Independently and a bit later, Giesl and coauthors have addressed the analysis of the lazy functional language Haskell [8].

Termination of typed λ-calculi. Quite a few people have written about termination based on types. Various subsets of the λ-calculus, in particular subsets typable by various disciplines, have been proven strongly normalising. Work in this direction includes pathbreaking results by Tait [24] and others concerning simple types, and Girard's System F [9]. Abel, Barthe and others have done newer type-based approaches to show termination of a λ-calculus extended with recursive data types [1, 2, 3].

Typed functional languages: Xi's Ph.D. research focused on tracing value flow via data types for termination verification in higher order programming languages [28], Wahlstedt

has an approach to combine size-change termination analysis with constructive type theory [26, 27].

Term rewriting systems: The popular "dependency pair" method was developed by Arts and Giesl [6] for first-order programs in TRS form. This community has begun to study termination of higher order term rewriting systems, including research by Giesl et.al. [7, 8], Toyama [25] and others.

2. The call-by-value λ-calculus

First, we review relevant definitions and results for the call-by-value λ-calculus, and then provide an observable characterisation of the behavior of a nonterminating expression.

2.1. Classical semantics.

Definition 2.1. *Exp* is the set of all λ-expressions that can be formed by these syntax rules, where `@` is the *application operator* (sometimes omitted). We use the `teletype` font for λ-expressions.

```
e, P   ::=   x | e @ e | λx.e
x      ::=   Variable name
```

- The set of *free variables* $fv(\mathtt{e})$ is defined as usual: $fv(\mathtt{x}) = \{\mathtt{x}\}$, $fv(\mathtt{e}@\mathtt{e}') = fv(\mathtt{e}) \cup fv(\mathtt{e}')$ and $fv(\lambda\mathtt{x}.\mathtt{e}) = fv(\mathtt{e}) \setminus \{\mathtt{x}\}$. A *closed* λ-expression `e` satisfies $fv(\mathtt{e}) = \emptyset$.
- A *program*, usually denoted by `P`, is any closed λ-expression.
- The set of *subexpressions* of a λ-expression `e` is denoted by $subexp(\mathtt{e})$.

The following is standard, e.g., [18]. Notation: β-reduction is done by substituting v for all free occurrences of `x` in `e`, written $\mathtt{e}[v/\mathtt{x}]$, and renaming λ-bound variables if needed to avoid capture.

Definition 2.2. (Call-by-value semantics) The *call-by-value evaluation relation* is defined by the following inference rules, with judgement form $\mathtt{e} \Downarrow v$ where `e` is a closed λ-expression and $v \in$ *ValueS*. *ValueS* (for "standard value") is the set of all abstractions $\lambda\mathtt{x}.\mathtt{e}$.

$$\frac{}{v \Downarrow v} \ \textit{If } v \in \textit{ValueS} \ \text{(ValueS)} \qquad \frac{\mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 \quad \mathtt{e}_2 \Downarrow v_2 \quad \mathtt{e}_0[v_2/\mathtt{x}] \Downarrow v}{\mathtt{e}_1@\mathtt{e}_2 \Downarrow v} \ \text{(ApplyS)}$$

Lemma 2.3. (Determinism) *If* $\mathtt{e} \Downarrow v$ *and* $\mathtt{e} \Downarrow w$ *then* $v = w$.

2.2. Nontermination is sequential.

A proof of $\mathtt{e} \Downarrow v$ is a finite object, and no such proof exists if the evaluation of `e` fails to terminate. Thus in order to be able to trace an arbitrary computation, terminating or not, we introduce a new "calls" relation $\mathtt{e} \rightarrow \mathtt{e}'$, in order to make nontermination visible.

The "calls" relation. The rationale is straightforward: $\mathtt{e} \to \mathtt{e}'$ if in order to deduce $\mathtt{e} \Downarrow v$ for some value v, it is necessary first to deduce $\mathtt{e}' \Downarrow u$ for some u, i.e., some inference rule has form $\dfrac{\ldots\ \mathtt{e}' \Downarrow ?\ \ldots}{\mathtt{e} \Downarrow ?}$. Applying this to Definition 2.2 gives the following.

Definition 2.4. (Evaluation and call semantics) The *evaluation and call relations* are defined by the following inference rules, where $\underset{r}{\to}$, $\underset{d}{\to}$, $\underset{c}{\to} \subseteq \mathit{Exp} \times \mathit{Exp}$[1].

$$\frac{}{v \Downarrow v}\ \mathit{If}\ v \in \mathit{ValueS}\ (\text{Value}) \qquad \frac{}{\mathtt{e}_1@\mathtt{e}_2 \underset{r}{\to} \mathtt{e}_1}\ (\text{Operator}) \qquad \frac{\mathtt{e}_1 \Downarrow v_1}{\mathtt{e}_1@\mathtt{e}_2 \underset{d}{\to} \mathtt{e}_2}\ (\text{Operand})$$

$$\frac{\mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 \quad \mathtt{e}_2 \Downarrow v_2}{\mathtt{e}_1@\mathtt{e}_2 \underset{c}{\to} \mathtt{e}_0[v_2/\mathtt{x}]}\ (\text{Call}_0) \qquad \frac{\mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 \quad \mathtt{e}_2 \Downarrow v_2 \quad \mathtt{e}_0[v_2/\mathtt{x}] \Downarrow v}{\mathtt{e}_1@\mathtt{e}_2 \Downarrow v}\ (\text{Apply}_0)$$

For convenience we will sometimes combine the three into a single *call relation* $\to = \underset{r}{\to} \cup \underset{d}{\to} \cup \underset{c}{\to}$. As usual, we write $\to^+$ for the transitive closure of $\to$, and $\to^*$ for its reflexive transitive closure. We will sometimes write $s \Downarrow$ to mean $s \Downarrow v$ for some $v \in \mathit{ValueS}$, and write $s \not\Downarrow$ to mean there is no $v \in \mathit{ValueS}$ such that $s \Downarrow v$, i.e., if evaluation of s does not terminate.

A small improvement to the operational semantics. Note that rules (Call_0) and (Apply_0) from Definition 2.4 overlap: $\mathtt{e}_2 \Downarrow v_2$ appears in both, as does $\mathtt{e}_0[v_2/\mathtt{x}]$. Thus ($\text{Call}_0$) can be used as an intermediate step to simplify (Apply_0), giving a more orthogonal set of rules. Variations on the following combined set will be used in the rest of the paper:

Definition 2.5. (Combined evaluate and call rules, standard semantics)

$$\frac{}{v \Downarrow v}\ \mathit{If}\ v \in \mathit{ValueS}\ (\text{Value}) \qquad \frac{}{\mathtt{e}_1@\mathtt{e}_2 \underset{r}{\to} \mathtt{e}_1}\ (\text{Operator}) \qquad \frac{\mathtt{e}_1 \Downarrow v_1}{\mathtt{e}_1@\mathtt{e}_2 \underset{d}{\to} \mathtt{e}_2}\ (\text{Operand})$$

$$\frac{\mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 \quad \mathtt{e}_2 \Downarrow v_2}{\mathtt{e}_1@\mathtt{e}_2 \underset{c}{\to} \mathtt{e}_0[v_2/\mathtt{x}]}\ (\text{Call}) \qquad \frac{\mathtt{e}_1@\mathtt{e}_2 \underset{c}{\to} \mathtt{e}' \quad \mathtt{e}' \Downarrow v}{\mathtt{e}_1@\mathtt{e}_2 \Downarrow v}\ (\text{Apply})$$

The *call tree* of program $\mathtt{P}$ is the smallest set of expressions CT containing $\mathtt{P}$ that is closed under $\to$. It is not necessarily finite.

Lemma 2.6. (*NIS, or* **N***ontermination* **I***s* **S***equential*) *Let* $\mathtt{P}$ *be a program. Then* $\mathtt{P} \Downarrow$ *if and only if CT has no infinite call chain starting with* $\mathtt{P}$*:*

$$\mathtt{P} = \mathtt{e}_0 \to \mathtt{e}_1 \to \mathtt{e}_2 \to \ldots$$

[1]Naming: r, d in $\underset{r}{\to}$, $\underset{d}{\to}$ are the last letters of *operato***r** and *operan***d**, and c in $\underset{c}{\to}$ stands for "call".

Example: evaluation of expression $\Omega = (\lambda \mathtt{x.x@x})@(\lambda \mathtt{y.y@y})$ yields an infinite call chain:

$$\Omega = (\lambda \mathtt{x.x@x})@(\lambda \mathtt{y.y@y}) \rightarrow (\lambda \mathtt{y.y@y})@(\lambda \mathtt{y.y@y}) \rightarrow (\lambda \mathtt{y.y@y})@(\lambda \mathtt{y.y@y}) \rightarrow \ldots$$

By the NIS Lemma all nonterminating computations give rise to *infinite linear call chains*. Such call chains need not, however, be repetitive as in this example, or even finite.

Informally $\mathtt{e}_0 \not\Downarrow$ implies existence of an infinite call chain as follows: Try to build, bottom-up and left-to-right, a proof tree for $\mathtt{e}_0 \Downarrow v$. Since call-by-value evaluation cannot "get stuck" this process will continue infinitely, leading to an infinite call chain. Figure 1 shows such a call tree with infinite path starting with $\mathtt{e}_0 \rightarrow \mathtt{e}_1 \rightarrow \mathtt{e}_2 \rightarrow \mathtt{e}_3 \rightarrow \ldots$, where $\rightarrow = \underset{r}{\rightarrow} \cup \underset{d}{\rightarrow} \cup \underset{c}{\rightarrow}$. The Appendix contains a formal proof.

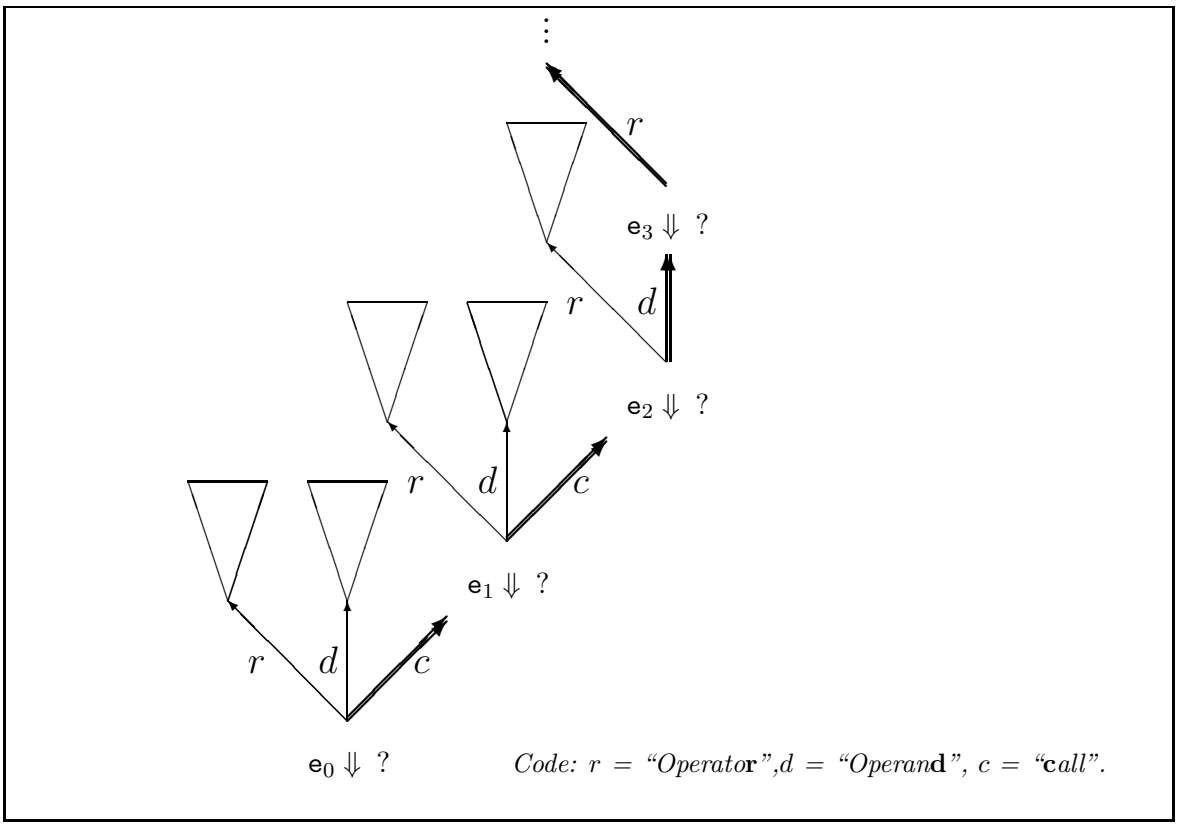

Figure 1: Nontermination implies existence of an infinite call chain

3. An approach to termination analysis

The "size-change termination" analysis of Lee, Jones and Ben-Amram [14] is based on several concepts, including:

(1) Identifying nontermination as caused by *infinitely long sequential state transitions*.
(2) A fixed set of *program control points*.
(3) *Observable decreases* in data value sizes.
(4) *Construction* of one size-change graph for each function call.
(5) Finding the program's *control flow graph*, and the call sequences that follow it.

The NIS Lemma establishes point 1. However, concepts 2, 3, 4 and 5 all seem a priori absent from the λ-calculus, except that an application must be a call; and even then, it is not a priori clear *which* function is being called. We will show, one step at a time, that all the concepts do in fact exist in call-by-value λ-calculus evaluation.

3.1. **An environment-based semantics.** Program flow analysis usually requires evident program control points. An alternate environment-based formulation remedies their absence in the λ-calculus. The ideas were formalised by Plotkin [18], and have long been used in implementations of functional programming language such as SCHEME and ML.

Definition 3.1. (States, etc.) Define *State*, *Value*, *Env* to be the smallest sets such that

$$\begin{array}{lcl} \mathit{State} & = & \{ \quad \mathtt{e} : \rho \quad | \quad \mathtt{e} \in \mathit{Exp}, \rho \in \mathit{Env} \text{ and } \mathit{dom}(\rho) \supseteq \mathit{fv}(\mathtt{e}) \quad \} \\ \mathit{Value} & = & \{ \quad \lambda\mathtt{x}.\mathtt{e} : \rho \quad | \quad \lambda\mathtt{x}.\mathtt{e} : \rho \in \mathit{State} \quad \} \\ \mathit{Env} & = & \{ \quad \rho : X \to \mathit{Value} \quad | \quad X \text{ is a finite set of variables} \quad \} \end{array}$$

Equality of states is defined by:

$$\mathtt{e}_1 : \rho_1 = \mathtt{e}_2 : \rho_2 \text{ holds if } \mathtt{e}_1 = \mathtt{e}_2 \text{ and } \rho_1(x) = \rho_2(x) \text{ for all } \mathtt{x} \in fv(\mathtt{e}_1)$$

The empty environment with domain $X = \emptyset$ is written []. The environment-based evaluation judgement form is $s \Downarrow v$ where $s \in \mathit{State}, v \in \mathit{Value}$.

The Plotkin-style rules follow the pattern of Definition 2.1, except that substitution (β-reduction) $\mathtt{e}_0[v_2/\mathtt{x}]$ of the (CallS) rule is replaced by a "lazy substitution" that just updates the environment in the new (Call) rule. Further, variable values are fetched from the environment

Definition 3.2. (Environment-based evaluation semantics) The evaluation relation $\Downarrow$, is defined by the following inference rules.

$$\frac{}{v \Downarrow v} \ \mathit{If}\ v \in \mathit{Value}\ \text{(ValueE)} \qquad \frac{}{\mathtt{x} : \rho \Downarrow \rho(\mathtt{x})}\ \text{(VarE)}$$

$$\frac{\mathtt{e}_1 : \rho \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0 \quad \mathtt{e}_2 : \rho \Downarrow v_2 \quad \mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2] \Downarrow v}{\mathtt{e}_1@\mathtt{e}_2 : \rho \Downarrow v}\ \text{(ApplyE}_0\text{)}$$

3.2. **States are tree structures.** A state has form $s = \mathtt{e} : \rho$ as in Definition 3.1 where ρ binds the free variables of $\mathtt{e}$ to values, which are themselves states. Consider, for two examples, these two states

$$s \quad = \quad \mathtt{e}{:}\rho \quad = \quad \mathtt{r@(r@a)}{:}[\mathtt{r} \mapsto \mathtt{succ} : [], \mathtt{a} \mapsto \underline{\mathtt{2}} : []]$$

$$s' \quad = \quad \mathtt{e'}{:}\rho' \quad = \quad \mathtt{r@(r@a)}{:}[\mathtt{r} \mapsto \lambda\mathtt{a}\,.\,\mathtt{r@(r@a)} : [\mathtt{r} \mapsto \mathtt{succ} : []], \mathtt{a} \mapsto \underline{\mathtt{2}} : []]$$

(written in our usual linear notation and using the standard Church numerals $\underline{0}, \underline{1}, \underline{2}, \ldots$. For brevity details of the successor function succ are omitted. It is straightforward to verify that $s \Downarrow \underline{4}$ and $s' \Downarrow \underline{6}$ by Definition 3.2.

More generally, each value bound in an environment is a state in turn, so in full detail a state's structure is a *finite tree*. (The levels of this tree represent variable bindings, not to be confused with the syntactic or subexpression tree structures from Figure 3.)

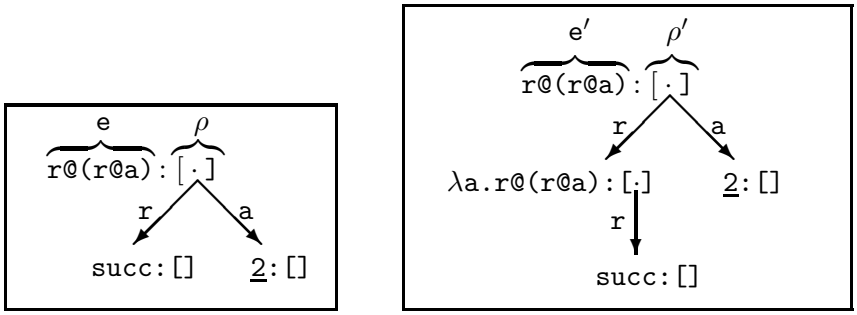

Figure 2: Structures of two states s, s'. Each state is a finite tree.

Figure 2 shows the structure of these two states, with abbreviations for Church numerals such as $\underline{2} = \lambda \mathtt{s} \lambda \mathtt{z} \,.\, \mathtt{s@(s@z)}$.

3.3. **Nontermination made visible in an environment-based semantics.** Straightforwardly adapting the approach of Section 2.2. gives the following set of inference rules, variations on which will be used in the rest of the paper:

Definition 3.3. (Combined evaluate and call rules, environment semantics)

$$\frac{}{v \Downarrow v} \ \textit{If } v \in \textit{Value} \ \text{(Value)} \qquad \frac{}{\mathtt{x} : \rho \Downarrow \rho(\mathtt{x})} \ \text{(Var)}$$

$$\frac{}{\mathtt{e}_1\mathtt{@}\mathtt{e}_2 : \rho \underset{r}{\rightarrow} \mathtt{e}_1 : \rho} \ \text{(Operator)} \qquad \frac{\mathtt{e}_1 : \rho \Downarrow v_1}{\mathtt{e}_1\mathtt{@}\mathtt{e}_2 : \rho \underset{d}{\rightarrow} \mathtt{e}_2 : \rho} \ \text{(Operand)}$$

$$\frac{\mathtt{e}_1 : \rho \Downarrow \lambda \mathtt{x}.\mathtt{e}_0 : \rho_0 \quad \mathtt{e}_2 : \rho \Downarrow v_2}{\mathtt{e}_1\mathtt{@}\mathtt{e}_2 : \rho \underset{c}{\rightarrow} \mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2]} \ \text{(Call)} \qquad \frac{\mathtt{e}_1\mathtt{@}\mathtt{e}_2 : \rho \underset{c}{\rightarrow} \mathtt{e}' : \rho' \quad \mathtt{e}' : \rho' \Downarrow v}{\mathtt{e}_1\mathtt{@}\mathtt{e}_2 : \rho \Downarrow v} \ \text{(Apply)}$$

The following is proven in the same way as Lemma 2.6.

Lemma 3.4. (*NIS, or* **N***ontermination* **I***s* **S***equential*) *Let* P *be a program. Then* $\mathtt{P} : [] \Downarrow$ *if and only if CT has no infinite call chain staring with* $\mathtt{P} : []$ *(where* $\rightarrow = \underset{r}{\rightarrow} \cup \underset{d}{\rightarrow} \cup \underset{c}{\rightarrow}$ *):*

$$\mathtt{P} : [] = \mathtt{e}_0 : \rho_0 \rightarrow \mathtt{e}_1 : \rho_1 \rightarrow \mathtt{e}_2 : \rho_2 \rightarrow \ldots$$

Following the lines of Plotkin [18], the environment-based semantics is shown equivalent to the usual semantics in the sense that they have the same termination behaviour. Further, when evaluation terminates the computed values are related by function $F : \mathit{States} \rightarrow \mathit{Exp}$ defined by

$$F(\mathtt{e} : \rho) = \mathtt{e}[F(\rho(\mathtt{x}_1))/\mathtt{x}_1, ..., F(\rho(\mathtt{x}_k))/\mathtt{x}_k] \ \text{ where } \{\mathtt{x}_1, .., \mathtt{x}_k\} = fv(\mathtt{e}).$$

Lemma 3.5. $\mathtt{P} : [] \Downarrow v$ *(by Definition 3.2) implies* $\mathtt{P} \Downarrow F(v)$ *(by Definition 2.5), and* $\mathtt{P} \Downarrow w$ *implies there exists* v' *such that* $\mathtt{P} : [] \Downarrow v'$ *and* $F(v') = w$.

Example: evaluation of closed $\Omega = (\lambda \mathtt{x}.\mathtt{x@x})@(\lambda \mathtt{y}.\mathtt{y@y})$ yields an infinite call chain:

$$\Omega : [] = (\lambda \mathtt{x}.\mathtt{x@x})@(\lambda \mathtt{y}.\mathtt{y@y}) : [] \to \mathtt{x@x} : \rho_1 \to \mathtt{y@y} : \rho_2 \to \mathtt{y@y} : \rho_2 \to \mathtt{y@y} : \rho_2 \to \ldots$$

where $\rho_1 = [\mathtt{x} \mapsto \lambda \mathtt{y}.\mathtt{y@y} : []]$ and $\rho_2 = [\mathtt{y} \mapsto \lambda \mathtt{y}.\mathtt{y@y} : []]$.

3.4. **A control point is a subexpression of a λ-expression.** The following *subexpression property* does not hold for the classical rewriting λ-calculus semantics, but does hold for Plotkin-style environment semantics of Definition 3.2. It is central to our program analysis: A *control point* will be a subexpression of the program P being analysed, and our analyses will trace program information flow to and from subexpressions of P.

Lemma 3.6. *If* $\mathtt{P} : [] \Downarrow \lambda \mathtt{x}.\mathtt{e} : \rho$ *then* $\lambda \mathtt{x}.\mathtt{e} \in \mathit{subexp}(\mathtt{P})$. [Recall Definition 2.1.]

This is proven as follows, using a more general inductive hypothesis.

Definition 3.7. The *expression support* of a given state s is $\mathit{exp\_sup}(s)$, defined by

$$\mathit{exp\_sup}(\mathtt{e} : \rho) = \mathit{subexp}(\mathtt{e}) \cup \bigcup_{\mathtt{x} \in \mathit{fv}(\mathtt{e})} \mathit{exp\_sup}(\rho(\mathtt{x}))$$

Lemma 3.8. (Subexpression property) *If* $s \Downarrow s'$ *or* $s \to s'$ *then* $\mathit{exp\_sup}(s) \supseteq \mathit{exp\_sup}(s')$.

Proof. This follows by induction on the proof of $s \Downarrow v$ or $s \to s'$. Lemma 3.6 is an immediate corollary.

Base cases: $s = \mathtt{x} : \rho$ and $s = \lambda \mathtt{x}.\mathtt{e} : \rho$ are immediate. For rule (Call) suppose $\mathtt{e}_1 : \rho \Downarrow \lambda \mathtt{x}.\mathtt{e}_0 : \rho_0$ and $\mathtt{e}_2 : \rho \Downarrow v_2$. By induction

$$\mathit{exp\_sup}(\mathtt{e}_1 : \rho) \supseteq \mathit{exp\_sup}(\lambda \mathtt{x}.\mathtt{e}_0 : \rho_0) \text{ and } \mathit{exp\_sup}(\mathtt{e}_2 : \rho) \supseteq \mathit{exp\_sup}(v_2)$$

Thus

$$\begin{array}{rcl} \mathit{exp\_sup}(\mathtt{e}_1@\mathtt{e}_2 : \rho) & \supseteq & \mathit{exp\_sup}(\mathtt{e}_1 : \rho) \cup \mathit{exp\_sup}(\mathtt{e}_2 : \rho) \quad \supseteq \\ \mathit{exp\_sup}(\lambda \mathtt{x}.\mathtt{e}_0 : \rho_0) \cup \mathit{exp\_sup}(v_2) & \supseteq & \mathit{exp\_sup}(\mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2]) \end{array}$$

For rule (Apply) we have $\mathit{exp\_sup}(\mathtt{e}_1@\mathtt{e}_2 : \rho) \supseteq \mathit{exp\_sup}(\mathtt{e}' : \rho') \supseteq \mathit{exp\_sup}(v)$. The cases (Operator), (Operand) are immediate. □

3.5. **Finitely describing a program's computation space.** A standard approach to program analysis is to trace data flow along the arcs of the program's *dynamic control graph* or DCG. In our case this is the call relation → of Definition 2.5. Unfortunately the DCG may be infinite, so for program analysis we will instead compute a safe finite approximation called the SCG, for *static control graph*.

Example 3.9. Figure 3 shows the combinator $\Omega = (\lambda \mathtt{x}.\mathtt{x@x})@(\lambda \mathtt{y}.\mathtt{y@y})$ as a syntax tree whose subexpressions are labeled by numbers. To its right is the "calls" relation →. It has an infinite call chain:

$$\Omega : [] \to \mathtt{x@x} : \rho_1 \to \mathtt{y@y} : \rho_2 \to \mathtt{y@y} : \rho_2 \to \mathtt{y@y} : \rho_2 \to \ldots$$

Using subexpression numbers, the loop is

$$\mathtt{1} : [] \to \mathtt{3} : \rho_1 \to \mathtt{7} : \rho_2 \to \mathtt{7} : \rho_2 \to \ldots$$

where $\rho_1 = [\mathtt{x} \mapsto \lambda \mathtt{y}.\mathtt{y@y} : []]$ and $\rho_2 = [\mathtt{y} \mapsto \lambda \mathtt{y}.\mathtt{y@y} : []]$. The set of states reachable from $\mathtt{P} : []$ is finite, so this computation is in fact a "repetitive loop." (It is also possible that a computation will reach infinitely many states that are all different.)

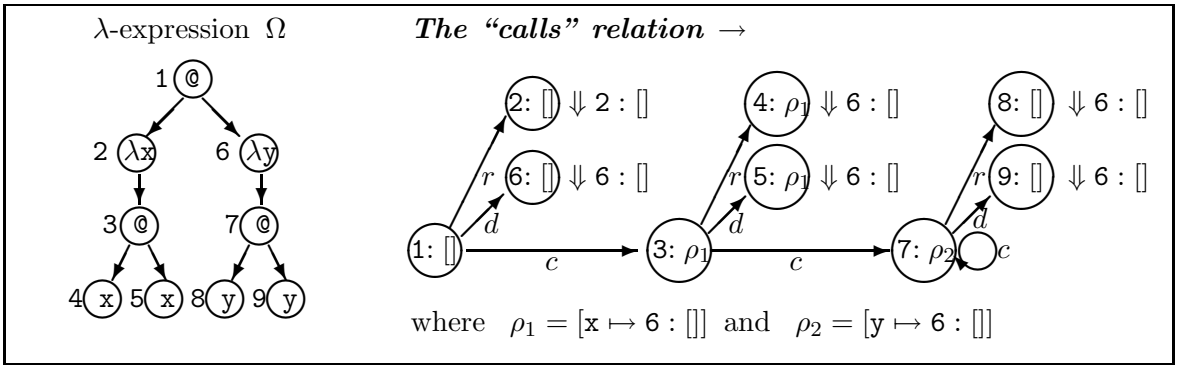

Figure 3: The DCG or dynamic control graph of a λ-expression

By the NIS Lemma 3.4, if $\mathtt{P} \not\Downarrow$ then there exists an infinite call chain

$$\mathtt{P} : [] = \mathtt{e}_0 : \rho_0 \rightarrow \mathtt{e}_1 : \rho_1 \rightarrow \mathtt{e}_2 : \rho_2 \rightarrow \ldots$$

By Lemma 3.8, $\mathtt{e}_i \in \mathit{subexp}(\mathtt{P})$ for each i. Our termination-detecting algorithm will focus on the *size relations between consecutive environments* ρ_i and ρ_{i+1} in this chain. Since $\mathit{subexp}(\mathtt{P})$ is a finite set, at least one subexpression $\mathtt{e}$ occurs infinitely often, so "self-loops" will be of particular interest.

Since all states have an expression component lying in a set of fixed size, and each expression in the environment also lies in this finite set, in an infinite state set $\mathcal{S}$ there will be states whose *environment depths* are arbitrarily large.

3.6. **Static control flow graphs for λ-expressions.** The end goal, given program P, is implied by the NIS Lemma 3.4: correctly to assert the nonexistence of any infinite call chain starting at $\mathtt{P} : []$. By the Subexpression Lemma 3.8 an infinite call chain $\mathtt{e}_0 : \rho_0 \rightarrow \mathtt{e}_1 : \rho_1 \rightarrow \mathtt{e}_2 : \rho_2 \rightarrow \ldots$ can only contain finitely many different expression components $\mathtt{e}_i$. A static control flow graph (SCG for short) including all expression components can be obtained by abstract interpretation of the "Calls" and "Evaluates-to" relations (Cousot and Cousot [4]). Figure 4 shows a SCG for Ω.

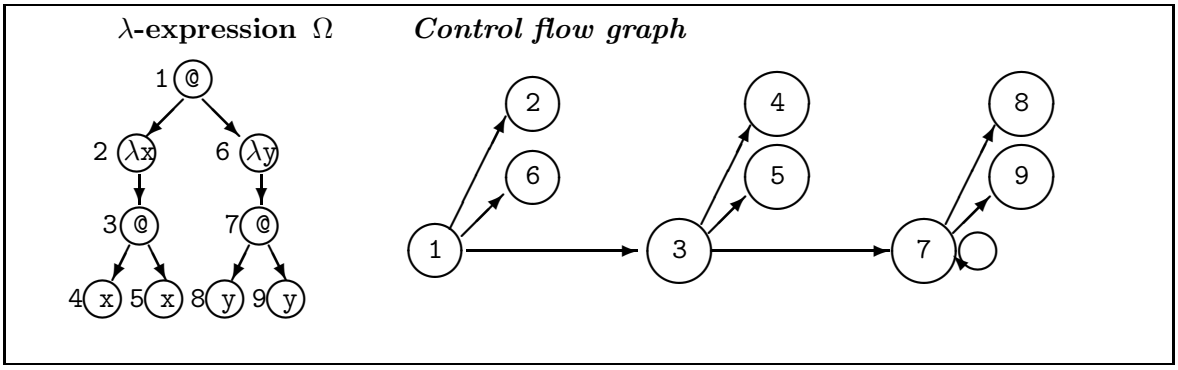

Figure 4: The SCG or static control graph of a λ-expression

An approximating SCG may be obtained by removing all environment components from Definition 3.3. To deal with the absence of environments the variable lookup rule is

modified: If $\mathtt{e}_1@\mathtt{e}_2$ is *any* application in P such that $\mathtt{e}_1$ can evaluate to a value of form $\lambda\mathtt{x}.\mathtt{e}$ and $\mathtt{e}_2$ can evaluate to value v_2, then v_2 is regarded as a possible value of x.

Although approximate, these rules have the virtue that there are only finitely many possible judgements $\mathtt{e} \to \mathtt{e}'$ and $\mathtt{e} \Downarrow \mathtt{e}'$. Consequently, the runtime behavior of program P may be (approximately) analysed by exhaustively applying these inference rules. A later section will extend the rules so they also generate size-change graphs.

Definition 3.10. (Approximate evaluation and call rules) The new judgement forms are $\mathtt{e} \Downarrow \mathtt{e}'$ and $\mathtt{e} \to \mathtt{e}'$. The inference rules are:

$$\frac{}{\lambda\mathtt{x}.\mathtt{e} \Downarrow \lambda\mathtt{x}.\mathtt{e}}\ \text{(ValueA)} \qquad \frac{\mathtt{e}_1@\mathtt{e}_2 \in \mathit{subexp}(\mathtt{P}) \quad \mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 \quad \mathtt{e}_2 \Downarrow v_2}{\mathtt{x} \Downarrow v_2}\ \text{(VarA)}$$

$$\frac{}{\mathtt{e}_1@\mathtt{e}_2 \underset{r}{\to} \mathtt{e}_1}\ \text{(OperatorA)} \qquad \frac{}{\mathtt{e}_1@\mathtt{e}_2 \underset{d}{\to} \mathtt{e}_2}\ \text{(OperandA)}$$

$$\frac{\mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 \quad \mathtt{e}_2 \Downarrow v_2}{\mathtt{e}_1@\mathtt{e}_2 \underset{c}{\to} \mathtt{e}_0}\ \text{(CallA)} \qquad \frac{\mathtt{e}_1@\mathtt{e}_2 \underset{c}{\to} \mathtt{e}' \quad \mathtt{e}' \Downarrow v}{\mathtt{e}_1@\mathtt{e}_2 \Downarrow v}\ \text{(ApplyA)}$$

The (VarA) rule refers globally to P, the program being analysed. The approximate evaluation is nondeterministic, since an expression may evaluate to more than one value.

Following is a central result: that all possible values obtained by the *actual evaluation* of Definition 3.3 are accounted for by the *approximate evaluation* of Definition 3.10.

Lemma 3.11.

$$\begin{array}{l} \textit{If } \mathtt{P} : [] \to^* \mathtt{e} : \rho \quad \textit{and} \quad \mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho', \quad \textit{then} \quad \mathtt{e} \Downarrow \mathtt{e}'. \\ \textit{If } \mathtt{P} : [] \to^* \mathtt{e} : \rho \quad \textit{and} \quad \mathtt{e} : \rho \to \mathtt{e}' : \rho', \quad \textit{then} \quad \mathtt{e} \to \mathtt{e}'. \end{array}$$

Proof is in the Appendix.

4. A quick review of size-change analysis

Using the framework of [14], the relation between two states s_1 and s_2 in a call $s_1 \to s_2$ or an evaluation $s_1 \Downarrow s_2$ will be described by means of a size-change graph G.

Example 4.1. Let first-order functions f and g be defined by mutual recursion:

```
f(x,y)   = if x=0 then y else 1: g(x,y,y)
g(u,v,w) = if w=0 then 3:f(u-1,w) else 2:g(u,v-1,w+2)
```

Label the three function calls 1, 2 and 3. The "control flow graph" in Figure 5 shows the calling function and called function of each call, e.g., $1 : \mathtt{f} \to \mathtt{g}$. Associate with each call a "size-change graph", e.g., G_1 for call 1, that safely describes the data flow from the calling function's parameters to the called function's parameters. Symbol $\downarrow$ indicates a value decrease.

Termination reasoning: We show that *all infinite size-change graph sequences* $\mathcal{M} = g_1 g_2 \ldots \in \{G_1, G_2, G_3\}^\omega$ that follow the program's control flow *are impossible* (assuming that the data value set is well-founded):

Case 1: $\mathcal{M} \in \ldots (G_2)^\omega$ ends in infinitely many G_2's: This would imply that *variable* v *descends infinitely.*

Case 2: $\mathcal{M} \in \ldots (G_1 G_2^* G_3)^\omega$. This would imply that *variable* u *descends infinitely.*

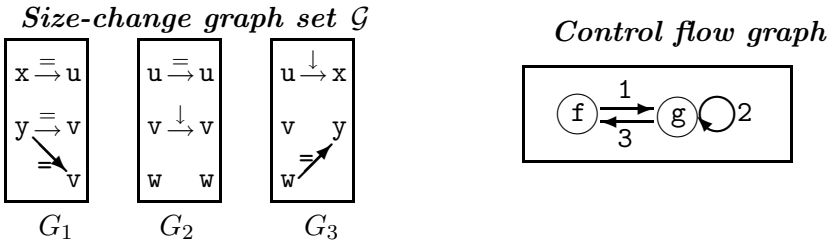

Figure 5: Call graph and size-change graphs for the example first-order program.

Both cases are impossible; therefore a call of *any program function with any data will terminate.* *End of example.*

Definition 4.2.

(1) A *size-change graph* $A \xrightarrow{G} B$ consists of a *source* set A; a *target* set B; and a set of labeled[2] arcs $G \subseteq A \times \{=, \downarrow\} \times B$.

(2) The *identity* size-change graph for A is $A \xrightarrow{id_A} A$ where $id_A = \{\mathtt{x} \xrightarrow{=} \mathtt{x} \mid \mathtt{x} \in A\}$.

(3) Size-change graphs $A \xrightarrow{G_1} B$ and $C \xrightarrow{G_2} D$ are *composible* if $B = C$. The *composition* of $A \xrightarrow{G_1} B$ and $B \xrightarrow{G_2} C$ is $A \xrightarrow{G_1;G_2} C$ where

$$\begin{aligned} G_1;G_2 = \quad & \{\mathtt{x} \xrightarrow{\downarrow} \mathtt{z} \mid \ \downarrow \ \in \{\, r,s \mid \mathtt{x} \xrightarrow{r} \mathtt{y} \in G_1 \text{ and } \mathtt{y} \xrightarrow{s} \mathtt{z} \in G_2 \text{ for some } \mathtt{y} \in B\} \ \} \\ \cup \quad & \{\mathtt{x} \xrightarrow{=} \mathtt{z} \mid \{=\} = \{\, r,s \mid \mathtt{x} \xrightarrow{r} \mathtt{y} \in G_1 \text{ and } \mathtt{y} \xrightarrow{s} \mathtt{z} \in G_2 \text{ for some } \mathtt{y} \in B\} \ \} \end{aligned}$$

Lemma 4.3. *Composition is associative, and* $A \xrightarrow{G} B$ *implies* $id_A; G = G; id_B = G$.

Definition 4.4. A *multipath* $\mathcal{M}$ over a set $\mathcal{G}$ of size-change graphs is a finite or infinite composible sequence of graphs in $\mathcal{G}$. Define

$$\mathcal{G}^\omega = \{\mathcal{M} = G_0, G_1, \dots \mid \text{graphs } G_i, G_{i+1} \text{ are composible for } i = 0, 1, 2, \dots \}$$

Definition 4.5.

(1) A *thread* in a multipath $\mathcal{M} = G_0, G_1, G_2, \dots$ is a sequence $t = a_j \xrightarrow{r_j} a_{j+1} \xrightarrow{r_{j+1}} \dots$ such that $a_k \xrightarrow{r_k} a_{k+1} \in G_k$ for every $k \geq j$ (and each r_k is $=$ or $\downarrow$.)

(2) Thread t is of *infinite descent* if $r_k = \downarrow$ for infinitely many $k \geq j$.

Definition 4.6. The size-change condition.

> A set $\mathcal{G}$ of size-change graphs satisfies the *size-change condition* if every infinite multipath $\mathcal{M} \in \mathcal{G}^\omega$ contains at least one thread of infinite descent.

Perhaps surprisingly, the size-change condition is decidable. Its worst-case complexity is shown to be complete for PSPACE in [14] (for first-order programs, in relation to the length of the program being analysed).

The example revisited The program of Figure 5 has three size-change graphs, one for each of the calls $\mathtt{1} : \mathtt{f} \to \mathtt{g}, \mathtt{2} : \mathtt{g} \to \mathtt{g}, \mathtt{3} : \mathtt{g} \to \mathtt{f}$, so $\mathcal{G} = \{A \xrightarrow{G_1} B, B \xrightarrow{G_2} B, B \xrightarrow{G_3} A\}$ where $A = \{\mathtt{x}, \mathtt{y}\}$ and $B = \{\mathtt{u}, \mathtt{v}, \mathtt{w}\}$. (Note: the vertical layout of size-change graphs in Figure 5 is inessential; one could simply write $G_3 = \{\mathtt{u} \xrightarrow{\downarrow} \mathtt{x}, \mathtt{w} \xrightarrow{=} \mathtt{y}\}$.)

$\mathcal{G}$ satisfies the size-change condition, since *every infinite multipath has either a thread that decreases* u *infinitely, or a thread that decreases* v *infinitely.*

[2]Arc label $\overline{\geq}$ signifying $\geq$ was used in [14] instead of $=$, but this makes no difference in our context.

5. Tracing data size changes in call-by-value λ-calculus evaluation

The next focus is on size relations between consecutive environments in a call chain.

5.1. Size changes in a computation: a well-founded relation between states.

Definition 5.1.

(1) A *name path* is a finite string p of variable names, where the empty string is (as usual) written ϵ.

(2) The *graph basis* of a state $s = \mathtt{e} : \rho$ is the smallest set $gb(s)$ of name paths satisfying

$$gb(\mathtt{e} :\rho) = \{\epsilon\} \cup \{\mathtt{x}p \mid \mathtt{x} \in fv(\mathtt{e}) \text{ and } \mathtt{p} \in gb(\rho(\mathtt{x}))\}$$

By this definition, for the two states in the example above we have $gb(s) = \{\epsilon, \mathtt{r}, \mathtt{a}\}$ and $gb(s') = \{\epsilon, \mathtt{r}, \mathtt{rr}, \mathtt{a}\}$. Further, given a state s and a path $p \in gb(s)$, we can find the substate identified by name path p as follows:

Definition 5.2. The *valuation function* $\overline{s} : gb(s) \to State$ of a state s is defined by:

$$\overline{s}(\epsilon) = s \text{ and } \overline{\mathtt{e} : \rho}(\mathtt{x}p) = \overline{\rho(\mathtt{x})}(p)$$

We need to develop a size ordering on states. This will be modeled by size-change arcs $\stackrel{=}{\to}$ and $\stackrel{\downarrow}{\to}$. The size relation we use is partly the "subtree" relation on closure values $\mathtt{e} : \rho$, and partly the "subexpression" relation on λ-expressions.

Definition 5.3.

(1) The *state support* of a state $\mathtt{e} : \rho$ is given by

$$support(\mathtt{e} : \rho) = \{\mathtt{e} : \rho\} \cup \bigcup_{\mathtt{x} \in fv(\mathtt{e})} support(\rho(\mathtt{x}))$$

(2) Relations $\succ_1$, $\succ_2$, $\succeq$ and $\succ$ on states are defined by:

- $s_1 \succ_1 s_2$ holds if $support(s_1) \ni s_2$ and $s_1 \neq s_2$;
- $s_1 \succ_2 s_2$ holds if $s_1 = \mathtt{e}_1 : \rho_1$ and $s_2 = \mathtt{e}_2 : \rho_2$, where $subexp(\mathtt{e}_1) \ni \mathtt{e}_2$ and $\mathtt{e}_1 \neq \mathtt{e}_2$ and $\forall \mathtt{x} \in fv(\mathtt{e}_2).\rho_1(x) = \rho_2(x)$. Further,

- Relation $\succeq$ is defined to be the transitive closure of $\succ_1 \cup \succ_2 \cup =$.
- Finally, $s_1 \succ s_2$ if $s_1 \succeq s_2$ and $s_1 \neq s_2$

Lemma 5.4. *The relation $\succ \subseteq State \times State$ is well-founded.*

We prove that the relation $\succ$ on states is well-founded by proving that

$$\mathtt{e}_1 : \rho_1 \succ \mathtt{e}_2 : \rho_2 \text{ implies that } (H(\mathtt{e}_1 : \rho_1), L(\mathtt{e}_1)) >_{lex} (H(\mathtt{e}_2 : \rho_2), L(\mathtt{e}_2))$$

in the lexicographic order, where H gives the height of the environment and L gives the length of the expression. The proof is in the Appendix.

Lemma 5.5. *If $p \in gb(s)$ then $s \succeq_1 \overline{s}(p)$. If $p \in gb(s)$ and $p \neq \epsilon$ then $s \succ_1 \overline{s}(p)$.*

6. Size-change graphs that safely describe a program

6.1. **Safely describing state transitions.** We now define the arcs of the size-change graphs (recalling Definition 4.2):

Definition 6.1. A size-change graph G relating state s_1 to state s_2 has *source* $gb(s_1)$ and *target* $gb(s_2)$.

Definition 6.2. Let $s_1 = \mathtt{e}_1 : \rho_1$ and $s_2 = \mathtt{e}_2 : \rho_2$. Size-change graph $\mathtt{s}_1{\rightarrow}\mathtt{s}_2, G$ is *safe*[3] for (s_1, s_2) if

$$p_1 \stackrel{=}{\rightarrow} p_2 \in G \;\text{ implies }\; \overline{s_1}(p_1) = \overline{s_2}(p_2) \;\text{ and }\; p_1 \stackrel{\downarrow}{\rightarrow} p_2 \in G \;\text{ implies }\; \overline{s_1}(p_1) \succ \overline{s_2}(p_2)$$

By $dom(G)$ we denote the subset of $source(G)$ from where arcs begin. By $codom(G)$ we denote the subset of $target(G)$ where arcs end. Notice that if a size-change graph G is safe for the states (s_1, s_2), then any subset size-change graph $G' \subset G$ with $source(G') = source(G)$ and $target(G') = target(G)$ is safe for (s_1, s_2).

Definition 6.3. A set $\mathcal{G}$ of size-change graphs is *safe for program* `P` if $\mathtt{P} : [] \;\rightarrow^*\; s_1 \rightarrow s_2$ implies some $G \in \mathcal{G}$ is safe for the pair (s_1, s_2).

Example 6.4. Figure 6 below shows a graph set $\mathcal{G}$ that is safe for the program $\Omega = (\lambda\mathtt{x}.\mathtt{x@x})(\lambda\mathtt{y}.\mathtt{y@y})$. For brevity, each subexpression of Ω is referred to by number in the diagram of $\mathcal{G}$. Subexpression $\mathtt{1} = \Omega$ has no free variables, so arcs from node `1` are labeled with size-change graphs $G_0 = \emptyset$.

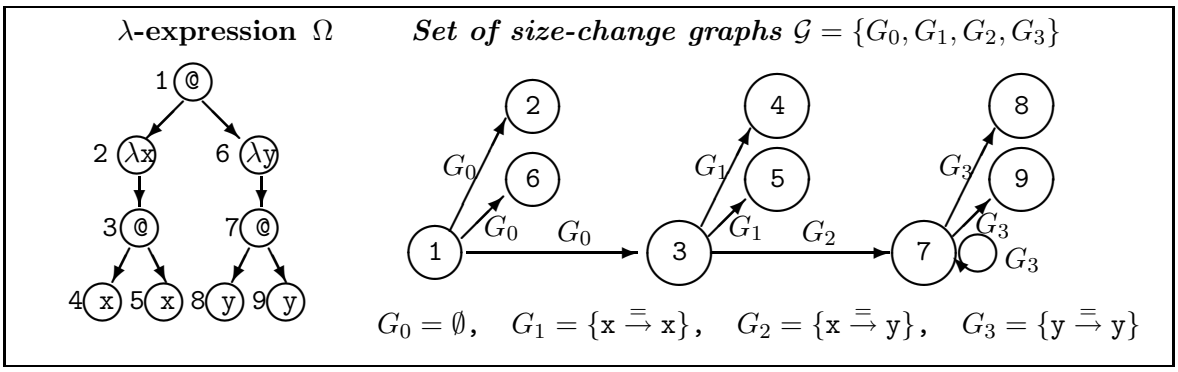

Figure 6: A set of size-change graphs that safely describe Ω's nonterminating computation

Theorem 6.5. *If $\mathcal{G}$ is safe for program* `P` *and satisfies the size-change condition, then call-by-value evaluation of* `P` *terminates.*

Proof. Suppose call-by-value-evaluation of `P` does not terminate. Then by Lemma 3.4 there is an infinite call chain

$$\mathtt{P} : [] = \mathtt{e}_0 : \rho_0 \rightarrow \mathtt{e}_1 : \rho_1 \rightarrow \mathtt{e}_2 : \rho_2 \rightarrow \ldots$$

Letting $s_i = \mathtt{e}_i : \rho_i$, by safety of $\mathcal{G}$ (Definition 6.3), there is a size-change graph $G_i \in \mathcal{G}$ that safely describes each pair (s_i, s_{i+1}). By the size-change condition (Definition 4.6) the multipath $\mathcal{M} = G_0, G_1, \ldots$ has an infinite thread $t = a_j \stackrel{r_j}{\rightarrow} a_{j+1} \stackrel{r_{j+1}}{\rightarrow} \ldots$ such that

[3]The term "safe" comes from abstract interpretation [13]. An alternative would be "sound."

$k \geq j$ implies $a_k \overset{r_k}{\rightarrow} a_{k+1} \in G_k$, and each r_k is $\downarrow$ or $=$, and there are infinitely many $r_k = \downarrow$. Consider the value sequence $\overline{s_j}(a_j), \overline{s_{j+1}}(a_{j+1}), \ldots$. By safety of G_k (Definition 6.2) we have $\overline{s_k}(a_k) \succeq \overline{s_{k+1}}(a_{k+1})$ for every $k \geq j$, and infinitely many proper decreases $\overline{s_k}(a_k) \succ \overline{s_{k+1}}(a_{k+1})$. However this is impossible since by Lemma 5.4 the relation $\succ$ on *State* is well-founded.

Conclusion: call-by-value-evaluation of P terminates. □

The goal is partly achieved: We have found a sufficient condition on a set of size-change graphs to guarantee program termination. What we have not yet done is to find an algorithm to *construct* a size-change graph set $\mathcal{G}$ that is safe for P (The safety condition of Definition 6.3 is in general undecidable, so enumeration of all graphs won't work.) Our graph construction algorithm is developed in two stages:

- First, the exact evaluation and call relations are "instrumented" so as to produce safe size-change graphs during evaluation.
- Second, an extension of the abstract interpretation from Section 3.6 yields a *computable* over-approximation $\mathcal{G}$ that contains all graphs that can be built during exact evaluation.

6.2. **Generating size-change graphs during a computation.** We now "instrument" the exact evaluation and call relations so as to produce safe size-change graphs during evaluation. In the definition of the size-change graphs x, y, z are variables, and p, q can be variables or ϵ, the empty path. Recall the valuation function for a state gives $\bar{s}(\epsilon) = s$, so in a sense ϵ is bound to the whole state.

Definition 6.6. (Evaluation and call with graph generation) The extended evaluation and call judgement forms are $\mathtt{e} : \rho \rightarrow \mathtt{e}' : \rho', G$ and $\mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho', G$, where $source(G) = \mathrm{fv}(\mathtt{e}) \cup \{\epsilon\}$ and $target(G) = \mathrm{fv}(\mathtt{e}') \cup \{\epsilon\}$. The inference rules are:

$$\frac{}{\lambda \mathtt{x}.\mathtt{e} : \rho \Downarrow \lambda \mathtt{x}.\mathtt{e} : \rho, id^{=}_{\lambda \mathtt{x}.\mathtt{e}}} \text{ (ValueG)}$$

$$\frac{}{\mathtt{e}_1 @ \mathtt{e}_2 : \rho \underset{r}{\rightarrow} \mathtt{e}_1 : \rho, id^{\downarrow}_{\mathtt{e}_1}} \text{ (OperatorG)} \qquad \frac{\mathtt{e}_1 : \rho \Downarrow v_1}{\mathtt{e}_1 @ \mathtt{e}_2 : \rho \underset{d}{\rightarrow} \mathtt{e}_2 : \rho, id^{\downarrow}_{\mathtt{e}_2}} \text{ (OperandG)}$$

$id^{=}_{e}$ stands for $\{\epsilon \overset{=}{\rightarrow} \epsilon\} \cup \{\mathtt{y} \overset{=}{\rightarrow} \mathtt{y} \mid \mathtt{y} \in fv(\mathtt{e})\}$

$id^{\downarrow}_{e}$ stands for $\{\epsilon \overset{\downarrow}{\rightarrow} \epsilon\} \cup \{\mathtt{y} \overset{=}{\rightarrow} \mathtt{y} \mid \mathtt{y} \in fv(\mathtt{e})\}$

An arc $\mathtt{y} \overset{=}{\rightarrow} \mathtt{y}$ express that the state bound to the variable y is the same in both sides, before and after the evaluation or call.

The ϵ "represent" the whole state. In the (ValueG) rule the state $\lambda \mathtt{x}.\mathtt{e} : \rho$ is the same in both sides and so there is an arc $\epsilon \overset{=}{\rightarrow} \epsilon$. In the (OperatorG) and (OperandG) rules the state is smaller in the right hand side because we go to a strict subexpression and possibly also restrict the environment ρ accordingly. So there are $\epsilon \overset{\downarrow}{\rightarrow} \epsilon$ arcs.

$$\frac{}{\mathtt{x} : \rho \Downarrow \rho(\mathtt{x}),\ \{\mathtt{x} \overset{\downarrow}{\rightarrow} \mathtt{y} \mid \mathtt{y} \in fv(\mathtt{e}')\ \} \cup \{\mathtt{x} \overset{=}{\rightarrow} \epsilon\}} \ \rho(\mathtt{x}) = \mathtt{e}' : \rho' \text{ (VarG)}$$

In the (VarG) rule the state on the right side is $\rho(x)$. This is the state which x is bound to in the environment in the left hand side, therefore we have an arc $\mathtt{x} \overset{=}{\rightarrow} \epsilon$. Suppose

$\rho(\mathtt{x}) = \mathtt{e}' : \rho'$ and $\mathtt{y} \in fv(\mathtt{e}')$. Then $\mathtt{y}$ is bound in ρ' and this binding is then a subtree of $\mathtt{e}' : \rho'$. So we have an arc $\mathtt{x} \xrightarrow{\downarrow} \mathtt{y}$.

$$\frac{\mathtt{e}_1 : \rho \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0, G_1 \quad \mathtt{e}_2 : \rho \Downarrow v_2, G_2}{\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{c}{\rightarrow} \mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2], G_1^{-\epsilon/\lambda x.e_0} \cup_{e_0} G_2^{\epsilon \mapsto \mathtt{x}}} \text{ (CallG)}$$

In the definition of the size-change graphs used in the (CallG) rule $\mathtt{x}$, $\mathtt{y}$, $\mathtt{z}$ are variables, and p, q can be variables or ϵ. In $\xrightarrow{r}$ the r can be either $\downarrow$ or $=$. The construction of the size-change graph associated with the call is explained below.

$G_1^{-\epsilon/\lambda x.e_0}$	stands for	cases
	$\mathtt{x} \in fv(\mathtt{e}_0)$	$: \{\mathtt{y} \xrightarrow{r} \mathtt{z} \mid \mathtt{y} \xrightarrow{r} \mathtt{z} \in G_1\} \cup \{\epsilon \xrightarrow{\downarrow} \mathtt{z} \mid \epsilon \xrightarrow{r} \mathtt{z} \in G_1\}$
	$\mathtt{x} \notin fv(\mathtt{e}_0)$	$: \{\mathtt{y} \xrightarrow{r} \mathtt{z} \mid \mathtt{y} \xrightarrow{r} \mathtt{z} \in G_1\} \cup \{\epsilon \xrightarrow{\downarrow} q \mid \epsilon \xrightarrow{r} q \in G_1\} \cup \{p \xrightarrow{\downarrow} \epsilon \mid p \xrightarrow{r} \epsilon \in G_1\}$
$G_2^{\epsilon \mapsto \mathtt{x}}$	stands for	$\{\ \mathtt{y} \xrightarrow{r} \mathtt{x} \mid \ \mathtt{y} \xrightarrow{r} \epsilon \in G_2\} \cup \{\ \epsilon \xrightarrow{\downarrow} \mathtt{x} \mid \ \epsilon \xrightarrow{r} \epsilon \in G_2\ \}$
$G \cup_e G'$	stands for	the restriction of $G \cup G'$ such that the codomain $\subseteq fv(\mathtt{e}) \cup \{\epsilon\}$

First we consider how much information from G_1 we can preserve. We have that the whole state $\mathtt{e}_1@\mathtt{e}_2 : \rho$ in left hand side for the c-call is strictly larger than $\mathtt{e}_1 : \rho$. The variable $\mathtt{x}$ is not free in $\lambda\mathtt{x}.\mathtt{e}_0$ and so does not belong to the target of G_1. If a variable $\mathtt{z} \in \mathrm{fv}(\lambda\mathtt{x}.\mathtt{e}_0)$ is bound in ρ_0 then it is bound to the same state in $\rho_0[\mathtt{x} \mapsto v_2]$. Therefore, if there is an arc $\mathtt{y} \xrightarrow{r} \mathtt{z}$ in G_1, then it also safely describes the c-call and can be preserved. Also, if there is an arc $\epsilon \xrightarrow{r} \mathtt{z}$ in G_1, then an arc $\epsilon \xrightarrow{\downarrow} \mathtt{z}$ describes the c-call. Further, if $\mathtt{x} \notin fv(\mathtt{e}_0)$ then $\mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2] = \mathtt{e}_0 : \rho_0$ and then $\lambda\mathtt{x}.\mathtt{e}_0 : \rho_0 \succ \mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2] = \mathtt{e}_0 : \rho_0$. In this case, if there is an arc $p \xrightarrow{r} \epsilon$ going to ϵ in G_1, then the arc $p \xrightarrow{\downarrow} \epsilon$ describes the c-call.
Now consider which information we can gain from G_2. We have that the whole state $\mathtt{e}_1@\mathtt{e}_2 : \rho$ in left hand side for the c-call is strictly larger than $\mathtt{e}_2 : \rho$. If $\mathtt{x} \in fv(\mathtt{e}_0)$ then in $\mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2]$ we have that $\mathtt{x}$ is bound to the whole state in the right hand side for the evaluation of the operand. So in this case, if there is an arc $\mathtt{y} \xrightarrow{r} \epsilon$ in G_2 then the arc $\mathtt{y} \xrightarrow{r} \mathtt{x}$ describes the c-call, and if there is an arc $\epsilon \xrightarrow{r} \epsilon$ in G_2 then the arc $\epsilon \xrightarrow{\downarrow} \mathtt{x}$ describes the c-call. If $\mathtt{x} \notin fv(\mathtt{e}_0)$ then we cannot gain any information from G_2. The restriction built into the definition of $\cup_{e_0}$ ensures that this holds.

$$\frac{\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{c}{\rightarrow} \mathtt{e}' : \rho', G' \quad \mathtt{e}' : \rho' \Downarrow v, G}{\mathtt{e}_1@\mathtt{e}_2 : \rho \Downarrow v, (G'; G)} \text{ (ApplyG)}$$

The size-change graph (G';G) is the composition of the two graphs.

In the size-change graphs generated by the rules above, the less-than relations ($x \xrightarrow{\downarrow} y$) in (VarG)-rule arise from the sub-environment property of $\succ_1$ from Lemma 5.5. The remaining relations $\xrightarrow{\downarrow}$ arise from the subexpression property of $\succ_2$. The relations based on the sub-environment property capture the case that the state on the right hand side is fetched from the environment in the left hand side. The equality relations $\xrightarrow{=}$ describe how values are preserved under calls and evaluations.

Lemma 6.7. *$s \to s'$ (by Definition 2.5) iff $s \to s', G$ (by Definition 6.6) for some G. Further, $s \Downarrow s'$ iff $s \Downarrow s', G$ for some G.*

Theorem 6.8. (The extracted graphs are safe)
$s \to s', G$ or $s \Downarrow s', G$ (by Definition 6.6) implies G is safe for (s, s') (with source and target sets extended as necessary).

Lemma 6.7 is immediate since the new rules extend the old, without any restriction on their applicability. Proof of "safety" Theorem 6.8 is in Appendix.

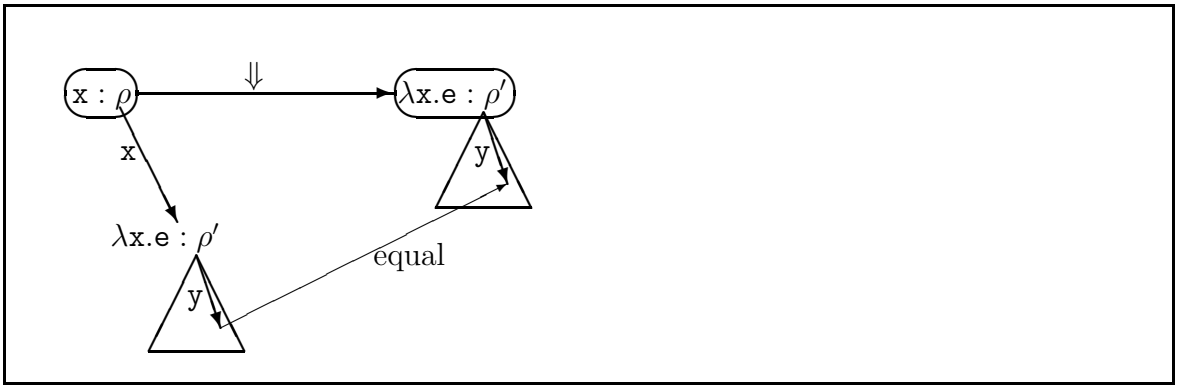

Figure 7: Data-flow in a variable evaluation

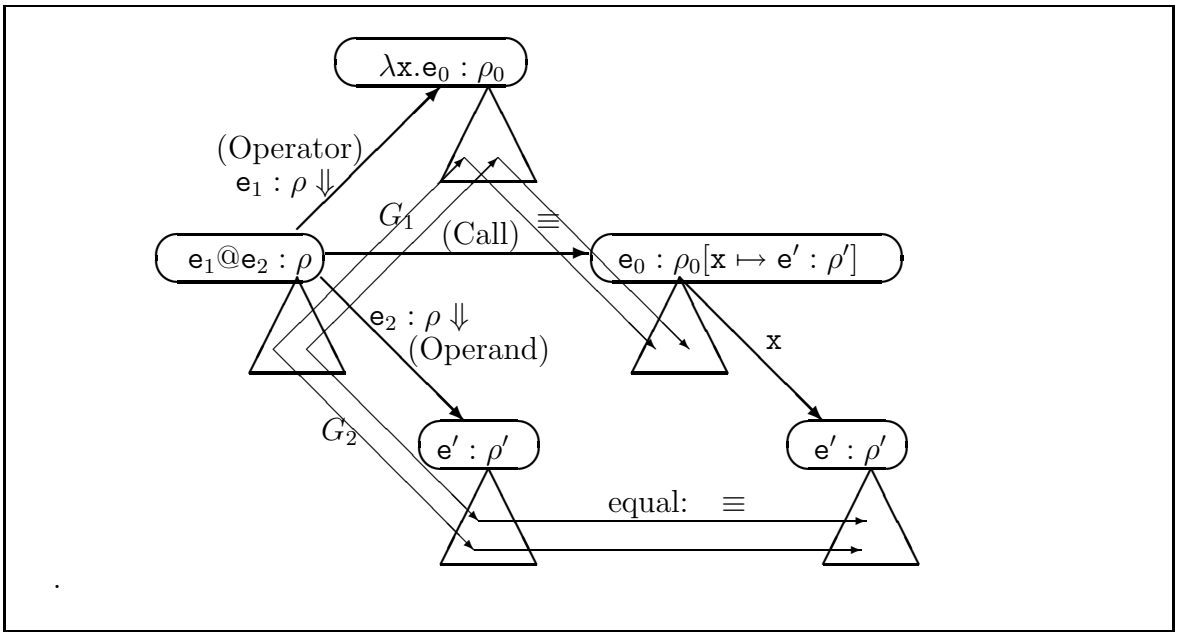

Figure 8: Data-flow in an application

The diagram of Figure 7 illustrates the data-flow in a variable evaluation. The diagram of Figure 8 may be of some use in visualising data-flow during evaluation of $\mathtt{e}_1@\mathtt{e}_2$. States are in ovals and triangles represent environments. In the application $\mathtt{e}_1@\mathtt{e}_2 : \rho$ on the left,

operator $\mathtt{e}_1 : \rho$ evaluates to $\lambda\mathtt{x}.\mathtt{e}_0 : \rho_0, G_1$ and operand $\mathtt{e}_2 : \rho$ evaluates to $\mathtt{e}' : \rho', G_2$. The size-change graphs G_1 and G_2 show relations between variables bound in their environments. There is a call from the application $\mathtt{e}_1@\mathtt{e}_2 : \rho$ to $\mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto \mathtt{e}' : \rho']$ the body of the operator-value with the environment extended with a binding of $\mathtt{x}$ to the operand-value $\mathtt{e}' : \rho'$.

It is possible to approximate the calls and evaluates to relations with different degrees of precision depending on how much information is kept about the bindings in the environment. Here we aim at a coarse approximation, where we remove all environment components.[4]

6.3. **Construction of size-change graphs by abstract interpretation.** We now extend the coarse approximation to construct size-change graphs.

Definition 6.9. (Approximate evaluation and call with graph generation)
The judgement forms are now $\mathtt{e} \to \mathtt{e}', G$ and $\mathtt{e} \Downarrow \mathtt{e}', G$, where source$(G) = \mathrm{fv}(\mathtt{e}) \cup \{\epsilon\}$ and target$(G) = \mathrm{fv}(\mathtt{e}') \cup \{\epsilon\}$. The inference rules are:

$$\frac{}{\lambda\mathtt{x}.\mathtt{e} \Downarrow \lambda\mathtt{x}.\mathtt{e}, id^{=}_{\lambda\mathtt{x}.\mathtt{e}}}\ (\text{ValueAG}) \qquad \frac{\mathtt{e}_1@\mathtt{e}_2 \in \mathit{subexp}(\mathtt{P})\ \ \mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0, G_1 \quad \mathtt{e}_2 \Downarrow v_2, G_2}{\mathtt{x} \Downarrow v_2,\ \{\mathtt{x} \xrightarrow{\downarrow} \mathtt{y} \mid \mathtt{y} \in \mathit{fv}(v_2)\} \cup \{\mathtt{x} \xrightarrow{=} \epsilon\}}\ (\text{VarAG})$$

$$\frac{}{\mathtt{e}_1@\mathtt{e}_2 \underset{r}{\to} \mathtt{e}_1, id^{\downarrow}_{\mathtt{e}_1}}\ (\text{OperatorAG}) \qquad \frac{}{\mathtt{e}_1@\mathtt{e}_2 \underset{d}{\to} \mathtt{e}_2, id^{\downarrow}_{\mathtt{e}_2}}\ (\text{OperandAG})$$

$$\frac{\mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0, G_1 \quad \mathtt{e}_2 \Downarrow v_2, G_2}{\mathtt{e}_1@\mathtt{e}_2 \underset{c}{\to} \mathtt{e}_0, G_1^{-\epsilon/\lambda x.e_0} \cup_{e_0} G_2^{\epsilon \mapsto \mathtt{x}}}\ (\text{CallAG}) \qquad \frac{\mathtt{e}_1@\mathtt{e}_2 \underset{c}{\to} \mathtt{e}', G' \quad \mathtt{e}' \Downarrow v, G}{\mathtt{e}_1@\mathtt{e}_2 \Downarrow v, G'; G}\ (\text{ApplyAG})$$

Lemma 6.10. *Suppose* $\mathtt{P} : [] \to^* \mathtt{e} : \rho$. *If* $\mathtt{e} : \rho \to \mathtt{e}' : \rho', G$ *by definition 6.6 then* $\mathtt{e} \to \mathtt{e}', G$. *Further, if* $\mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho', G$ *then* $\mathtt{e} \Downarrow \mathtt{e}', G$.

Proof. Follows from Lemma 3.11; see the Appendix. □

Definition 6.11.

$\mathit{absint}(\mathtt{P}) = \{\ G_j \mid j > 0 \wedge \exists \mathtt{e}_i, G_i (0 \le i \le j) : \mathtt{P} = \mathtt{e}_0 \wedge (\mathtt{e}_0 \to \mathtt{e}_1, G_1) \wedge \ldots \wedge (\mathtt{e}_{j-1} \to \mathtt{e}_j, G_j)\ \}$

Theorem 6.12.

(1) *The set* $\mathit{absint}(\mathtt{P})$ *is safe for* $\mathtt{P}$.
(2) *The set* $\mathit{absint}(\mathtt{P})$ *can be effectively computed from* $\mathtt{P}$.

Proof. Part 1: Suppose $\mathtt{P} : [] = s_0 \to s_1 \to \ldots \to s_j$. Theorem 6.8 implies $s_i \to s_{i+1}, G_i$ where each G_i is safe for the pair (s_i, s_{i+1}). Let $s_i = \mathtt{e}_i : \rho_i$. By Lemma 6.10, $\mathtt{e}_i \to \mathtt{e}_{i+1}, G_i$. By the definition of $\mathit{absint}(\mathtt{P})$, $G_j \in \mathit{absint}(\mathtt{P})$.

Part 2: There is only a fixed number of subexpressions of $\mathtt{P}$, or of possible size-change graphs with source and target $\subseteq \{\epsilon\} \cup \{\mathtt{x} \mid \mathtt{x}$ is a variable in $\mathtt{P}\ \}$. Thus $\mathit{absint}(\mathtt{P})$ can be computed by applying Definition 6.9 exhaustively, starting with $\mathtt{P}$, until no new graphs or subexpressions are obtained. □

[4]It is possible to keep a little more information in the graphs than we do here even with no knowledge about value-bindings in the environment. We have chosen the given presentation for simplicity.

7. Some examples

7.1. **A simple example.** Using Church numerals ($\mathtt{n} = \lambda\mathtt{s}\lambda\mathtt{z}.\mathtt{s}^n(\mathtt{z})$), we expect `2 succ 0` to reduce to `succ(succ 0)`. However this contains unreduced redexes because call-by-value does not reduce under a λ, so we force the computation to carry on through by applying `2 succ 0` to the identity (twice). This gives:

```
2 succ 0 id1 id2 where
  succ = λm.λs.λz. m s (s z)
  id1  = λx.x
  id2  = λy.y
```

After writing this out in full as a λ-expression, our analyser yields (syntactically sugared):

```
     [λs2.λz2.(s2 @ (s2 @ z2))]          -- two --
@    [λm.λs.λz.  15: ((m@s)@(s@z))]      -- succ --
@    [λs1.λz1.z1]                        -- zero --
@    [λx.x]                              -- id1 --
@    [λy.y]                              -- id2 --

Output of loops from an analysis of this program:

15→* 15: [(m,>,m),(s,=,s),(z,=,z)], []

Size-Change Termination: Yes
```

The first number refers to the program point, then comes a list of edges. The loop occurs because application of 2 forces the code for the successor function to be executed twice, with decreasing argument values `m`. The notation for edges is a little different from previously, here `(m,>,m)` stands for $\mathtt{m} \overset{\downarrow}{\rightarrow} \mathtt{m}$.

7.2. $fnx = x + 2^n$ **by Church numerals.** This more interesting program computes $fnx = x + 2^n$ by higher-order primitive recursion. If `n` is a Church numeral then expression `n g x` reduces to $\mathtt{g}^n(\mathtt{x})$. Let `x` be the successor function, and `g` be a "double application" functional. Expressed in a readable named combinator form, we get:

```
f n x    where
f n   =  if n=0 then succ else g(f(n-1))
g r a =  r(ra)
```

As a lambda-expression (applied to values $\mathtt{n} = 3, \mathtt{x} = 4$) this can be written:

```
[λn.λx.  n                                          -- n --
         @  [λr.λa. 11: (r@ 13: (r@a))]             -- g --
         @  [λ k.λ s.λ z.(s@((k@s)@z))]             - succ-
         @  x ]                                     -- x --

@           [λs2.λz2.  (s2@(s2@(s2@z2))) ]         -- 3 --
@           [λs1.λz1.  (s1@(s1@(s1@(s1@z1))))]     -- 4 --
```

Following is the output from program analysis. The analysis found the following loops from a program point to itself with the associated size-change graph and path. The first number refers to the program point, then comes a list of edges and last a list of numbers, the other program points that the loop passes through.

```
SELF Size-Change Graphs, no repetition of graphs:

11 →* 11: [(r,>,r)]           []
11 →* 11: [(a,=,a),(r,>,r)]   [13]
13 →* 13: [(a,=,a),(r,>,r)]   [11]
13 →* 13: [(r,>,r)]           [11,11]

Size-Change Termination: Yes
```

7.3. **Ackermann's function, second-order.** This can be written without recursion using Church numerals as: `a m n` where `a = λm.  m b succ` and `b = λg.λn.  n g (g 1)`. Consequently `a m = b`$^{\mathtt{m}}$`(succ)` and `b g n =` $g^{n+1}(1)$, which can be seen to agree with the usual first-order definition of Ackermann's function. Following is the same as a lambda-expression applied to argument values `m=2, n=3`, with numeric labels on some subexpressions.

```
(λm.m b succ) 2 3  =  (λm.m@b@succ)@2@3
(λm.m@(λg.λn.n@g@(g@1))@succ)@2@3
(λm.m@(λg.λn.[9:](n@g@[13:](g@1)))@succ)@2@3
 where
 1   =  λs1.λz1. [17:](s1@z1)
succ =  λk.λs.λz. [23:](s@[25:](k@s@z))
 2   =  λs2.λz2. s2@(s2@z2)
 3   =  λs3.λz3. [39:](s3@[41:](s3@[43:](s3@z3)))
```

Output from an analysis of this program is shown here.
(It is not always the case that the same loop is shown for all program points in its path)

```
SELF Size-Change Graphs, no repetition of graphs:

 9 →*  9: [(ϵ,>,n),(g,>,g)]          [13]
 9 →*  9: [(g,>,g)]                  [17]
13 →* 13: [(g,>,g)]                  [9]
17 →* 17: [(s1,>,s1)]                [9]
23 →* 23: [(k,>,k),(s,=,s),(z,=,z)]  [25]
23 →* 23: [(s,>,s)]                  [9]
23 →* 23: [(s,>,s),(z,>,k)]          [25,17,9]
25 →* 25: [(k,>,k),(s,=,s),(z,=,z)]  [23]
25 →* 25: [(s,>,s),(z,>,k)]          [17,9,23]
25 →* 25: [(s,>,s)]                  [23,9,23]
39 →* 39: [(s3,>,s3)]                [9]
41 →* 41: [(s3,>,s3)]                [9,39]
43 →* 43: [(s3,>,s3)]                [9,39,41]

Size-Change Termination: Yes
```

7.4. **Arbitrary natural numbers as inputs.** The astute reader may have noticed a limitation in the above examples: each only concerns *a single λ-expression*, e.g., Ackermann's function applied to argument values `m=2, n=3`.

In an implemented version of the λ-termination analysis a program may have an arbitrary natural number as input; this is represented by •. Further, programs can have as constants the predecessor, successor and zero-test functions, and if-then-else expressions. We show, by some examples using •, that the size-change termination approach can handle the Y-combinator.

In Section 8 we show how to do size-change analysis of λ-expressions applied to *sets of argument values* in a more classic context, using Church or other numeral notations instead of •.

7.5. **A minimum function, with general recursion and Y-combinator.** This program computes the minimum of its two inputs using the call-by-value combinator $Y = \lambda$`p. [λq.p@(λs.q@q@s)] @ [λt.p@(λu.t@t@u)]`. The program, first as a first-order recursive definition.

```
m x y = if x=0 then 0 else if y=0 then 0 else succ (m (pred x) (pred y))
```

Now, in λ-expression form for analysis.

```
{λp.  [λq.p@(λs.q@q@s)] @ [λt.p@(λu.t@t@u)]}  -- the Y combinator --
@
[λm.λx.λy.27: if  ((ztst @  x),
                   0,
                   32:  if  ((ztst @ y),
                             0,
                             37: succ @ 39: m @ (pred@x) @ (pred@y)]
@ •
@ •
```

```
Output of loops from an analysis of this program:

27 →* 27: [(x,>,x),(y,>,y)]    [32,37,39]
32 →* 32: [(x,>,x),(y,>,y)]    [37,39,27]
37 →* 37: [(x,>,x),(y,>,y)]    [39,27,32]
39 →* 39: [(x,>,x),(y,>,y)]    [27,32,37]

Size-Change Termination: Yes
```

7.6. **Ackermann's function, second-order with constants and Y-combinator.** Ackermann's function can be written as: `a m n` where `a m = b`$^{\texttt{m}}$`(suc)` and `b g n =` $\texttt{g}^{\texttt{n}+1}(1)$. The following program expresses the computations of both `a` and `b` by loops, using the Y combinator (twice).

```
[λ y.λ y1.
(y1 @
λ a.λ m.  11:if(  (ztst@m),
                  λ v.(suc@v),
                  19:(  (y @
                        λ b.λ f.λ n.
                        25:if(  (ztst@n),
                                29:(f@1),
                                32:f@34:b @ f @ (pred@n))
                  @ 41: a @ (pred@m)                        ]

@ {λp.[λq.p@(λs.  q@q@s)] @ [λt.p@(λu.t@t@u)]}
@ {λp1.[λq1.p1@(λs.72:q1@1q@s1)] @ [λt1.p1@(λu1.81:t1@t1@u1)]}
@ •
@ •
```

```
Output of loops from an analysis of this program:

SELF Size-Change Graphs no repetition of graphs:


11 →* 11: [(a,>,y),(m,>,m)]    [19,41,72]
11 →* 11: [(m,>,m)]            [19,41,72,11,19,41,72]
19 →* 19: [(a,>,y),(m,>,m)]    [41,72,11]
19 →* 19: [(m,>,m)]            [41,72,11,19,41,72,11]
25 →* 25: [(f,>,b),(f,>,f)]    [29]
25 →* 25: [(f,=,f),(n,>,n)]    [32,34]
25 →* 25: [(f,>,f)]            [29,25,32,34]
29 →* 29: [(f,>,f)]            [25]
32 →* 32: [(f,>,b),(f,>,f)]    [25]
32 →* 32: [(f,=,f),(n,>,n)]    [34,25]
32 →* 32: [(f,>,f)]            [25,32,34,25]
34 →* 34: [(f,=,f),(n,>,n)]    [25,32]
34 →* 34: [(f,>,b),(f,>,f)]    [25,29,25,32]
34 →* 34: [(f,>,f)]            [25,29,25,32,34,25,32]
41 →* 41: [(m,>,m)]            [72,11,19]
72 →* 72: [(s1,>,s1)]          [11,19,41]
81 →* 81: [(u1,>,u1)]          [11,19,41]

Size-Change Termination: Yes
```

7.7. **Imprecision of abstract interpretation.** It is natural to wonder whether the gross approximation of Definition 3.10 comes at a cost. The (VarA) rule can in effect "mix up" different function applications, losing the coordination between operator and operand that is present in the exact semantics.

We have observed this in practice: The first time we had programmed Ackermann's using explicit recursion, we used the same instance of Y-combinator for both loops, so the single Y-combinator expression was "shared". The analysis did not discover that the program terminated.

However when this was replaced by the "unshared" version above, with two instances of the Y-combinator (`y` and `y1`) (one for each application), the problem disappeared and termination was correctly recognised.

7.8. **A counterexample to a conjecture.** Sereni disproved in [20, 21] our **conjecture** that the size-change method would recognise as terminating any simply typed λ-expression. The root of the problem is the imprecision of abstract interpretation just noted. A counter-example: the λ-expression

$$E = (\lambda a.a(\lambda b.a(\lambda cd.d)))(\lambda e.e(\lambda f.f))$$

is simply-typable but not size-change terminating. Its types are any instantiation of

$$\begin{array}{lcl} a & : & ((\tau \to \tau) \to \mu \to \mu) \to \mu \to \mu \\ b, c & : & \tau \to \tau \\ d & : & \mu \\ e & : & (\tau \to \tau) \to \mu \to \mu \\ f & : & \tau \end{array}$$

8. Arbitrary λ-regular program inputs (Extended λ-calculus)

Above we have analysed the termination behaviour of a single closed λ-expression. We now analyse the termination behaviour for a program in the λ-calculus for all possible inputs from a given input-set of λ-expressions (e.g., Church numerals). The first step is to define which sets of λ-expressions we consider. A well-defined input set will be the set of closed expressions in the "language" generated by a λ-regular grammar.

We extend the syntax and semantics of the λ-calculus to handle expressions containing nonterminals. An extended lambda term represents all instances of a program with input taken from the input set. If our analysis certifies that the extended term terminates, then this implies that the program will terminate for all possible inputs.

8.1. **λ-regular grammars.** We are interested in a λ-regular grammar for the sake of the *language* that it generates: a set of pure λ-expressions (without nonterminals). This is done using the derivation relation $\Rightarrow^*_\Gamma$, soon to be defined.

Definition 8.1.

(1) A λ-*regular grammar* has form $\Gamma = (N, \Pi)$ where N is a finite set of *nonterminal* symbols and Π is a finite set of *productions*.

(2) A Γ-*extended* λ-*expression* has the following syntax:

`e, P`	::=	`x | A | e @ e | λx.e`
`A`	::=	Non-terminal name, $\mathtt{A} \in N$
`x`	::=	Variable name

Exp_Γ denotes the set of Γ-extended λ-expressions. Exp denotes the set of pure λ-expressions (without nonterminals). Clearly $Exp_\Gamma \supseteq Exp$.

(3) A production has form `A ::= e` where `e` is a Γ-extended λ-expression.

Definition 8.2. Let $\mathtt{nt(e)} = \{\mathtt{X}_1, \ldots, \mathtt{X}_k\}$ denote the *multi-set* of nonterminal occurrences in $\mathtt{e} \in Exp_\Gamma$. The *derivation relation* $\Rightarrow^*_\Gamma \subseteq Exp_\Gamma \times Exp$ is the smallest relation such that

(1) If $\mathtt{nt(e)} = \{\mathtt{X}_1, \ldots, \mathtt{X}_k\}$ and $\mathtt{X}_i \Rightarrow^*_\Gamma \mathtt{t}_i \in Exp$ for $i = 1, \ldots, k$, then $\mathtt{e} \Rightarrow^*_\Gamma \mathtt{e}[\mathtt{t}_1/\mathtt{X}_1, \ldots, \mathtt{t}_k/\mathtt{X}_k]$
(2) If $\mathtt{A} ::= \mathtt{e} \in \Gamma$ and $\mathtt{e} \Rightarrow^*_\Gamma \mathtt{e}'$ then $\mathtt{A} \Rightarrow^*_\Gamma \mathtt{e}'$.

Notice that $\Rightarrow^*_\Gamma$ relates extended λ-terms to pure λ-terms.

In the above definition 8.2 $\mathtt{nt(e)} = \{\mathtt{X}_1, \ldots, \mathtt{X}_k\}$ denotes the multi-set of nonterminals in $\mathtt{e}$ so two different $\mathtt{X}_i, \mathtt{X}_j$ may be instances of the same nonterminal $\mathtt{A}$. In the substitution $\mathtt{e}[\mathtt{t}_1/\mathtt{X}_1, \ldots, \mathtt{t}_k/\mathtt{X}_k]$ such two different instances of a nonterminal may be replaced by different pure λ-terms.

Example 8.3. A grammar for Church Numerals: Consider

$$\Gamma = (\{\mathtt{C}, \mathtt{A}\}, \{\mathtt{C} ::= \lambda\,\mathtt{s}\lambda\,\mathtt{z}\,.\,\mathtt{A},\ \mathtt{A} ::= \mathtt{z},\ \mathtt{A} ::= \mathtt{s}\,@\,\mathtt{A}\}$$

Here $\mathtt{A} \Rightarrow^*_\Gamma \mathtt{v}$ iff $\mathtt{v}$ has form $\mathtt{s}^n(\mathtt{z})$ for some $n \geq 0$. Clearly $\mathtt{C} \Rightarrow^*_\Gamma \mathtt{v}$ iff $\mathtt{v}$ has form $\lambda\,\mathtt{s}\lambda\,\mathtt{z}\,.\,\mathtt{s}^n(\mathtt{z})$ for some $n \geq 0$.

The following assumption makes proofs more convenient; proof is standard and so omitted.

Lemma 8.4. *For any λ-regular grammar Γ_0 there exists an equivalent λ-regular grammar Γ_1 such that no production in Γ_1 has form $\mathtt{A} ::= \mathtt{A}'$ where $\mathtt{A}' \in N$. We henceforth assume that all productions in a λ-regular grammar have form $\mathtt{A} ::= \mathtt{e}$ where $\mathtt{e} \notin N$.* □

Definition 8.5. In the following $\mathtt{e}$ is a Γ-extended λ-expression:

(1) Define the *free variables of* $\mathtt{e}$ by $fv(\mathtt{e}) = \{\mathtt{x} \mid \exists \mathtt{t}.\mathtt{e} \Rightarrow^*_\Gamma \mathtt{t}$ and $\mathtt{x} \in fv(\mathtt{t})\}$
(2) Define that $\mathtt{e}$ is *closed* iff t is closed for all t such that $e \Rightarrow^*_\Gamma t$. It follows that $\mathtt{e}$ is closed iff $fv(\mathtt{e}) = \{\}$.
(3) Define $subterms(\mathtt{e})$ inductively by:
  For a variable $\mathtt{x}$: $subterms(\mathtt{x}) = \{\mathtt{x}\}$.
  For an abstraction $\lambda\mathtt{x}.\mathtt{e}$: $subterms(\lambda\mathtt{x}.\mathtt{e}) = \{\lambda\mathtt{x}.\mathtt{e}\} \cup subterms(\mathtt{e})$.
  For an application $\mathtt{e}_1@\mathtt{e}_2$: $subterms(\mathtt{e}_1@\mathtt{e}_2) = \{\mathtt{e}_1@\mathtt{e}_2\} \cup subterms(\mathtt{e}_1) \cup subterms(\mathtt{e}_2)$.
  For a nonterminal $\mathtt{A}$: $subterms(\mathtt{A}) = \{\mathtt{A}\}$.
(4) Define $subexps(\mathtt{e})$ as the smallest set satisfying:
  For a variable $\mathtt{x}$: $subexps(\mathtt{x}) = \{\mathtt{x}\}$.
  For an abstraction $\lambda\mathtt{x}.\mathtt{e}$: $subexps(\lambda\mathtt{x}.\mathtt{e}) = \{\lambda\mathtt{x}.\mathtt{e}\} \cup subexps(\mathtt{e})$.
  For an application $\mathtt{e}_1@\mathtt{e}_2$: $subexps(\mathtt{e}_1@\mathtt{e}_2) = \{\mathtt{e}_1@\mathtt{e}_2\} \cup subexps(\mathtt{e}_1) \cup subexps(\mathtt{e}_2)$.
  For a nonterminal $\mathtt{A}$: $subexps(\mathtt{A}) = \{\mathtt{A}\} \cup \{\mathtt{t} \mid \exists \mathtt{e}.\mathtt{A} ::= \mathtt{e} \in \Gamma$ and $\mathtt{t} \in subexps(\mathtt{e})\}$.

If $\mathtt{e}' \in subterms(\mathtt{e})$ then $\mathtt{e}'$ is syntactically present as part of $\mathtt{e}$.
If $\mathtt{e}' \in subexps(\mathtt{e})$ then $\mathtt{e}'$ is either a subterm of $\mathtt{e}$ or a subexpression of a nonterminal $\mathtt{A} \in subterms(\mathtt{e})$.

Sets $subterms(\mathtt{e}), subexps(\mathtt{e})$ are both finite, and $subterms(\mathtt{e}) = subexps(\mathtt{e})$ for expressions $\mathtt{e}$ in the pure λ-calculus.

Example 8.6. In the grammar for Church Numerals $\mathtt{C}$ is a closed Γ-extended expression, but $\mathtt{A}$ is not a closed Γ-extended expression. Further, $subexps(\mathtt{A}) = \{\mathtt{A},\ \mathtt{z},\ \mathtt{s@A},\ \mathtt{s}\}$, $subexps(\mathtt{C}) = \{\mathtt{C},\ \lambda\,\mathtt{s}\lambda\,\mathtt{z}\,.\,\mathtt{A},\ \lambda\,\mathtt{z}\,.\,\mathtt{A},\ \mathtt{A},\ \mathtt{z},\ \mathtt{s@A},\ \mathtt{s}\}$, $fv(\mathtt{C}) = \{\}$, $fv(\mathtt{A}) = \{\mathtt{s}, \mathtt{z}\}$

Lemma 8.7. *Let $\mathtt{x}$ be a variable. If $\mathtt{A} \Rightarrow^*_\Gamma \mathtt{x}$ then $\mathtt{A} ::= \mathtt{x} \in \Gamma$.*
If $\mathtt{A} \Rightarrow^*_\Gamma \lambda\mathtt{x}.\mathtt{e}$ then there exists $\mathtt{e}' \in Exp_\Gamma$ such that $\mathtt{A} ::= \lambda\mathtt{x}.\mathtt{e}' \in \Gamma$.
If $\mathtt{A} \Rightarrow^*_\Gamma \mathtt{e}_1@\mathtt{e}_2$ then there exist $\mathtt{e}'_1, \mathtt{e}'_2 \in Exp_\Gamma$ such that $\mathtt{A} ::= \mathtt{e}'_1@\mathtt{e}'_2 \in \Gamma$.

Any production has one of the forms $\mathtt{A} ::= \mathtt{x}$, $\mathtt{A} ::= \lambda\mathtt{x}.\mathtt{e}$, $\mathtt{A} ::= \mathtt{e}_1@\mathtt{e}_2$. No production performed on a subterm (which must be a nonterminal) can give a new outermost syntactic term-constructor.

The following Lemma follows from the definition of free variables of an extended expression.

Lemma 8.8. *For a variable* $\mathtt{x}$*:* $fv(\mathtt{x}) = \{\mathtt{x}\}$.
For an abstraction $\lambda\mathtt{x}.\mathtt{e}$*:* $fv(\lambda\mathtt{x}.\mathtt{e}) = fv(\mathtt{e}) \setminus \{\mathtt{x}\}$.
For an application $\mathtt{e}_1@\mathtt{e}_2$*:* $fv(\mathtt{e}_1@\mathtt{e}_2) = fv(\mathtt{e}_1) \cup fv(\mathtt{e}_2)$.
For a nonterminal $\mathtt{A} \in N$*:* $fv(\mathtt{A}) = \{\mathtt{x} \mid \exists \mathtt{t}.\mathtt{A} \Rightarrow_\Gamma^* \mathtt{t} \text{ and } \mathtt{x} \in fv(\mathtt{t})\}$.

Lemma 8.9. *For* $\mathtt{A} \in N$ *the sets subexps*$(\mathtt{A})$ *and* $fv(\mathtt{A})$ *are finite and computable.*

Proof is straightforward.

8.2. **Extended environment-based semantics.** A semantics extending Definition 3.3 addresses the problem of substitution in expressions with non-terminals. Environments bind λ-variables (and not non-terminals) to values.

Definition 8.10. (Extended states, values and environments) *State*, *Value*, *Env* are the smallest sets such that

$$\begin{array}{lcl} \mathit{State} & = & \{ \quad \mathtt{e} : \rho \quad \mid \quad \mathtt{e} \in \mathit{Exp}_\Gamma, \rho \in \mathit{Env} \text{ and } fv(\mathtt{e}) \subseteq \mathit{dom}(\rho) \quad \} \\ \mathit{Value} & = & \{ \quad \lambda\mathtt{x}.\mathtt{e} : \rho \quad \mid \quad \lambda\mathtt{x}.\mathtt{e} : \rho \in \mathit{State} \quad \} \\ \mathit{Env} & = & \{ \quad \rho : X \to \mathit{Value} \quad \mid \quad X \text{ is a finite set of variables} \quad \} \end{array}$$

The empty environment with domain $X = \emptyset$ is written []. The evaluation judgement form is $s \Downarrow v$ where $s \in \mathit{State}, v \in \mathit{Value}$.

The following rules for calls and evaluations in the extended language are simple extensions of the rules for pure λ-calculus to also handle nonterminals.

Definition 8.11. (Extended environment-based evaluation) The judgement forms are $\mathtt{e} : \rho \to \mathtt{e}' : \rho'$ and $\mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho'$, where $\mathtt{e}, \mathtt{e}' \in \mathit{Exp}_\Gamma$, $e : \rho$ and $e' : \rho'$ are states. The evaluation and call relations $\Downarrow, \to$ are defined by the following inference rules, where $\to = \underset{r}{\to} \cup \underset{d}{\to} \cup \underset{c}{\to} \cup \underset{n}{\to}$.

$$\frac{}{A : \rho \underset{n}{\to} \mathtt{e} : \rho} \; A ::= \mathtt{e} \in \Gamma \text{ (GramX)} \qquad \text{New rule}$$

$$\frac{\mathtt{e} : \rho \underset{x}{\to} \mathtt{e}' : \rho' \quad \mathtt{e}' : \rho' \Downarrow v}{\mathtt{e} : \rho \Downarrow v} \; x \in \{c,n\} \text{ (ResultX)} \qquad \text{Extended } \textbf{Def.} \text{ 3.3 (Apply)}$$

The following rules have not been changed (but now expressions belong to Exp_Γ).

$$\frac{}{\lambda\mathtt{x}.\mathtt{e} : \rho \Downarrow \lambda\mathtt{x}.\mathtt{e} : \rho} \text{ (ValueX)} \qquad \frac{}{\mathtt{x} : \rho \Downarrow \mathtt{e}' : \rho'} \; \rho(\mathtt{x}) = \mathtt{e}' : \rho' \text{ (VarX)}$$

$$\frac{}{\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{r}{\to} \mathtt{e}_1 : \rho} \text{ (OperatorX)} \qquad \frac{\mathtt{e}_1 : \rho \Downarrow v_1}{\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{d}{\to} \mathtt{e}_2 : \rho} \text{ (OperandX)}$$

$$\frac{\mathtt{e}_1 : \rho \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0 \quad \mathtt{e}_2 : \rho \Downarrow v_2}{\mathtt{e}_1@\mathtt{e}_2 \underset{c}{\to} \mathtt{e}_0 : \rho_0[x \mapsto v_2]} \text{ (CallX)}$$

A Γ-extended *program* is a closed expression $\mathtt{P} \in Exp_\Gamma$. While evaluating a program in the extended language $(\mathtt{P} : [] \Downarrow \_)$, all calls and subevaluations will be from state to state.

In pure λ-calculus the evaluation relation is deterministic. The extended language is nondeterministic since a nonterminal $\mathtt{A}$ may have $\mathtt{A} ::= \mathtt{e}$ for more than one $\mathtt{e}$.

Informally explained, consider closed extended λ-expression $\mathtt{e@B}$ where nonterminal $\mathtt{B}$ satisfies $fv(\mathtt{B}) = \{\}$. Then $\mathtt{e@B}$ represents application of $\mathtt{e}$ to all possible inputs generated by $\mathtt{B}$. The analysis developed below can safely determine that $\mathtt{e}$ terminates on all inputs by analysing $\mathtt{e@B}$.

If a program in the extended language takes more than one input at a time, then we may rename the nonterminals and bound variables similarly as in α-conversion. As an example, if a program takes two Church numerals as input, then they can be given by two grammars identical in structure:

$\mathtt{C}_1 ::= \lambda\mathtt{s}_1.\lambda\mathtt{z}_1.\mathtt{A}_1 \quad \mathtt{A}_1 ::= \mathtt{z}_1 \quad \mathtt{A}_1 ::= \mathtt{s}_1@\mathtt{A}_1$ and

$\mathtt{C}_2 ::= \lambda\mathtt{s}_2.\lambda\mathtt{z}_2.\mathtt{A}_2 \quad \mathtt{A}_2 ::= \mathtt{z}_2 \quad \mathtt{A}_2 ::= \mathtt{s}_2@\mathtt{A}_2$

and we can analyse the termination behaviour for $(\mathtt{e@C}_1)@\mathtt{C}_2$. Such renaming can sometimes make the termination analysis more precise.

Definition 8.12. Suppose $\mathtt{e}$ is a closed Γ-extended expression and $nt(\mathtt{e}) = \{\mathtt{A}_1, \ldots, \mathtt{A}_k\}$ where $\Gamma = (N, \Pi)$ is a λ-regular grammar. By definition $\mathtt{e}$ is Γ-*terminating* iff

$$\mathtt{e}[\mathtt{t}_1/\mathtt{A}_1, \ldots, \mathtt{t}_k/\mathtt{A}_k] : [] \Downarrow$$

for all pure λ-expressions $\mathtt{t}_1, \ldots, \mathtt{t}_k$ such that $\mathtt{A}_i \Rightarrow_\Gamma^* \mathtt{t}_i$ for $i = 1, \ldots, k$.

The following rules for calls and evaluations with size-change graphs in the extended language are simple extensions of the rules for pure λ-calculus to also handle nonterminals.

Definition 8.13. (Environment-based evaluation and call semantics utilizing size-change graphs) The judgement forms are $\mathtt{e} : \rho \to \mathtt{e}' : \rho', G$ and $\mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho', G$, where $\mathtt{e}, \mathtt{e}' \in Exp_\Gamma$, $e : \rho$ and $e' : \rho'$ are states, $source(G) = fv(e) \cup \{\epsilon\}$ and $target(G) = fv(e') \cup \{\epsilon\}$. The evaluation and call relations $\Downarrow, \to$ are defined by the following inference rules, where $\to = \underset{r}{\to} \cup \underset{d}{\to} \cup \underset{c}{\to} \cup \underset{n}{\to}$.

$$\frac{}{A : \rho \underset{n}{\to} \mathtt{e} : \rho, id_e^{=}} \; A ::= \mathtt{e} \in \Gamma \text{ (GramG)} \qquad \text{New rule}$$

$$\frac{\mathtt{e} : \rho \underset{x}{\to} \mathtt{e}' : \rho', G' \quad \mathtt{e}' : \rho' \Downarrow v, G}{\mathtt{e} : \rho \Downarrow v, G'; G} \; x \in \{c,n\} \text{ (ResultG)} \qquad \text{Extended } \textbf{Def.}\ 6.6 \text{ (ApplyG)}$$

The following rules have not been changed (but now expressions belong to Exp_Γ).

$$\frac{}{\lambda\mathtt{x}.\mathtt{e} : \rho \Downarrow \lambda\mathtt{x}.\mathtt{e} : \rho, id^{=}_{\lambda\mathtt{x}.\mathtt{e}}} \text{ (ValueG)}$$

$$\frac{}{\mathtt{x} : \rho \Downarrow e' : \rho', \{\mathtt{x} \overset{=}{\to} \epsilon\} \cup \{\mathtt{x} \overset{\downarrow}{\to} \mathtt{y} \mid \mathtt{y} \in fv(e')\}} \; \rho(\mathtt{x}) = \mathtt{e}' : \rho' \text{ (VarG)}$$

$$\frac{}{\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{r}{\to} \mathtt{e}_1 : \rho, id^{\downarrow}_{\mathtt{e}_1}} \text{ (OperatorG)} \qquad \frac{\mathtt{e}_1 : \rho \Downarrow v_1}{\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{d}{\to} \mathtt{e}_2 : \rho, id^{\downarrow}_{\mathtt{e}_2}} \text{ (OperandG)}$$

$$\frac{\mathtt{e}_1 : \rho \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0, G_1 \quad \mathtt{e}_2 : \rho \Downarrow v_2, G_2}{\mathtt{e}_1@\mathtt{e}_2 \underset{c}{\rightarrow} \mathtt{e}_0 : \rho_0[x \mapsto v_2], G_1^{-\epsilon/\lambda x.e_0} \cup_{e_0} G_2^{\epsilon \mapsto \mathtt{x}}} \text{ (CallG)}$$

Theorem 8.14. (The extracted graphs are safe) *$s \to s', G$ or $s \Downarrow s', G$ implies G is safe for (s, s').*

Proof. This is shown by a case analysis as in the pure λ-calculus. For the (GramG) rule it is immediate from the definition of free variables for non-terminals. □

8.3. **Relating extended and pure λ-calculus.**
The aim is now to show that execution of a program P in the extended language can simulate execution of any program Q in the pure λ-calculus, where Q is derived from P by replacing each nonterminal occurrence A in P with a pure λ-expression A can produce. The converse does not hold: it is possible that there are simulated executions that do not correspond to any instantiated program Q. We have however certified a number of programs to terminate when applied to arbitrary Church numerals. An example is given at the end of this section.

Properties of the relation $\Rightarrow^*_\Gamma$
$\Rightarrow^*_\Gamma$ relates expressions $\mathtt{e}' \in Exp_\Gamma$ in the extended language to expressions $\mathtt{e} \in Exp$ in the pure lambda-calculus. Notice that there are only the following possible forms of $\Rightarrow^*_\Gamma$-related expressions:

$$\begin{array}{lll} \mathtt{x} \Rightarrow^*_\Gamma \mathtt{x} & \lambda\mathtt{x}.\mathtt{e}' \Rightarrow^*_\Gamma \lambda\mathtt{x}.\mathtt{e} & \mathtt{e}'_1@\mathtt{e}'_2 \Rightarrow^*_\Gamma \mathtt{e}_1@\mathtt{e}_2 \\ \mathtt{A} \Rightarrow^*_\Gamma \mathtt{x} & \mathtt{A} \Rightarrow^*_\Gamma \lambda\mathtt{x}.\mathtt{e} & \mathtt{A} \Rightarrow^*_\Gamma \mathtt{e}_1@\mathtt{e}_2 \end{array}$$

The relation $\Rightarrow^*_\Gamma$ has the following inductive properties:

$\mathtt{A} \Rightarrow^*_\Gamma \mathtt{t}$, for $\mathtt{A} \in N$ is given by definition 8.2.
$\mathtt{x} \Rightarrow^*_\Gamma \mathtt{x}$, – a variable x corresponds to the same variable x and nothing else.
$\lambda\mathtt{x}.\mathtt{e}' \Rightarrow^*_\Gamma \lambda\mathtt{x}.\mathtt{e}$, iff $\mathtt{e}' \Rightarrow^*_\Gamma \mathtt{e}$, same x.
$\mathtt{e}'_1@\mathtt{e}'_2 \Rightarrow^*_\Gamma \mathtt{e}_1@\mathtt{e}_2$ iff $\mathtt{e}'_1 \Rightarrow^*_\Gamma \mathtt{e}_1$ and $\mathtt{e}'_2 \Rightarrow^*_\Gamma \mathtt{e}_2$.

Lemma 8.15. *If $\mathtt{e}' \Rightarrow^*_\Gamma \mathtt{e}$ then $fv(\mathtt{e}') \supseteq fv(\mathtt{e})$.*

Proof. This is by induction on the structure of $\mathtt{e}'$.
Case $\mathtt{x} \Rightarrow^*_\Gamma \mathtt{x}$, immediate.
Case $\mathtt{A} \Rightarrow^*_\Gamma \mathtt{t}$ where $\mathtt{A} \in N$. By definition $fv(\mathtt{A}) = \{\mathtt{x} | \exists \mathtt{t}.\mathtt{A} \Rightarrow^*_\Gamma \mathtt{t} \text{ and } \mathtt{x} \in fv(\mathtt{t})\}$.
Case $\lambda\mathtt{x}.\mathtt{e}' \Rightarrow^*_\Gamma \lambda\mathtt{x}.\mathtt{e}$, iff $\mathtt{e}' \Rightarrow^*_\Gamma \mathtt{e}$. By induction the lemma holds for $\mathtt{e}'$ and e. Therefore $fv(\lambda\mathtt{x}.\mathtt{e}') = fv(\mathtt{e}') \setminus \{\mathtt{x}\} \supseteq fv(\mathtt{e}) \setminus \{\mathtt{x}\} = fv(\lambda\mathtt{x}.\mathtt{e})$.
Case $\mathtt{e}'_1@\mathtt{e}'_2 \Rightarrow^*_\Gamma \mathtt{e}_1@\mathtt{e}_2$, iff $\mathtt{e}'_1 \Rightarrow^*_\Gamma \mathtt{e}_1$ and $\mathtt{e}'_2 \Rightarrow^*_\Gamma \mathtt{e}_2$. By induction the lemma holds for $\mathtt{e}'_1, \mathtt{e}_1$ and $\mathtt{e}'_2, \mathtt{e}_2$. Hence $fv(\mathtt{e}'_1@\mathtt{e}'_2) = fv(\mathtt{e}'_1) \cup fv(\mathtt{e}'_2) \supseteq fv(\mathtt{e}_1) \cup fv(\mathtt{e}_2) = fv(\mathtt{e}_1@\mathtt{e}_2)$. □

If $\mathtt{e} \in Exp$, i.e., no nonterminals occur in $\mathtt{e}$, then $\mathtt{e} \Rightarrow^*_\Gamma \mathtt{e}$. If $\mathtt{A} \Rightarrow^*_\Gamma \mathtt{e}$ then there exist $\mathtt{t} \notin N$ such that $\mathtt{A} ::= \mathtt{t}$ and $\mathtt{t} \Rightarrow^*_\Gamma \mathtt{e}$.

Definition 8.16. The relation S between states
Define the relation S between states in the extended language and states in the pure λ-calculus as the smallest relation S such that:

$$S(\mathtt{e}' : \rho', \mathtt{e} : \rho) \text{ if } \mathtt{e}' \Rightarrow^*_\Gamma \mathtt{e} \text{ and for all } \mathtt{x} \in fv(\mathtt{e}) \text{ it holds that } S(\rho'(\mathtt{x}), \rho(\mathtt{x})).$$

If $\mathtt{e} : \rho$ is a state in the pure lambda calculus then it is also a state in the extended language and $S(\mathtt{e} : \rho, \mathtt{e} : \rho)$.

Lemma 8.17. *If $S(\mathtt{A} : \rho', \mathtt{e} : \rho)$ and $\mathtt{A} ::= \mathtt{t},\ \mathtt{t} \Rightarrow^*_\Gamma \mathtt{e}$ then also $S(\mathtt{t} : \rho', \mathtt{e} : \rho)$.* □

We now define a relation T between size-change graphs. The intention is that $T(G', G)$ is to hold when the only difference in the generation of the graphs is due to nonterminals that take the place of pure lambda expressions.

Definition 8.18. The relation T between size-change graphs
Define $T(G', G)$ to hold iff

i) $source(G') \supseteq source(G)$ and $target(G') \supseteq target(G)$.
ii) The subgraph of G' restricted to $source(G)$ and $target(G)$ is a subset of G.
iii) Furthermore if $\mathtt{z} \in source(G') \setminus source(G)$ then either there is no edge from $\mathtt{z}$ in G' or the only edge from $\mathtt{z}$ in G' is $(\mathtt{z} \stackrel{=}{\rightarrow} \mathtt{z})$, and if $(\mathtt{z} \stackrel{=}{\rightarrow} \mathtt{z}) \in G'$ then $\mathtt{z} \notin target(G)$.

We have that $T(G'_0, G_0)$, $T(G'_1, G_1)$, $\text{target}(G'_0) = \text{source}(G'_1)$ and $\text{target}(G_0) = \text{source}(G_1)$ together imply that $T((G'_0; G'_1), (G_0; G_1))$ holds.

Lemma 8.19. Simulation Property

i) If $S(\mathtt{e}' : \rho', \mathtt{e} : \rho)$ and $\mathtt{e} : \rho \Downarrow \mathtt{e}_0 : \rho_0, G$ then there exist $\mathtt{e}'_0 : \rho'_0, G'$ with $S(\mathtt{e}'_0 : \rho'_0, \mathtt{e}_0 : \rho_0)$ and $T(G', G)$ such that $\mathtt{e}' : \rho' \Downarrow \mathtt{e}'_0 : \rho'_0, G'$.
ii) If $S(\mathtt{e}' : \rho', \mathtt{e} : \rho)$ and $\mathtt{e} : \rho \underset{x}{\rightarrow} \mathtt{e}_0 : \rho_0, G$ with $x \in \{r, d, c\}$ then there exist $\mathtt{e}'_0 : \rho'_0, G'$ and possibly s such that either $\mathtt{e}' : \rho' \underset{x}{\rightarrow} \mathtt{e}'_0 : \rho'_0, G'$ or $\mathtt{e}' : \rho' \underset{n}{\rightarrow} s \underset{x}{\rightarrow} \mathtt{e}'_0 : \rho'_0, G'$ with $S(\mathtt{e}'_0 : \rho'_0, \mathtt{e}_0 : \rho_0)$, $T(G', G)$, and in the last case $S(s, \mathtt{e} : \rho)$.
The composite size-change graph for the double-call $\mathtt{e}' : \rho' \underset{n}{\rightarrow} s \underset{x}{\rightarrow} \mathtt{e}'_0 : \rho'_0$ will have the same edges as G' because the $\underset{n}{\rightarrow}$ call generates an $id^=$ graph.

Corollary 8.20. *For programs $\mathtt{P} \in Exp_\Gamma$ and $\mathtt{Q} \in Exp$ with $\mathtt{P} \Rightarrow^*_\Gamma \mathtt{Q}$ it holds that:*
If $\mathtt{Q} : [] \rightarrow^* \mathtt{e} : \rho$ then there exists $\mathtt{e}' : \rho'$ such that $\mathtt{P} : [] \rightarrow^* \mathtt{e}' : \rho'$ and $S(\mathtt{e}' : \rho', \mathtt{e} : \rho)$.
If $\mathtt{Q} : [] \Downarrow \mathtt{e} : \rho$ then there exist $\mathtt{e}' : \rho'$ such that $\mathtt{P} : [] \Downarrow \mathtt{e}' : \rho'$ and $S(\mathtt{e}' : \rho', \mathtt{e} : \rho)$.

Also notice that if $\mathtt{e}_1 : \rho_1 \underset{n}{\rightarrow} \mathtt{e}_2 : \rho_2$ then $fv(\mathtt{e}_1) \supseteq fv(\mathtt{e}_2)$ by the definition of free variables for nonterminals. (By definition, $S(\mathtt{e}' : \rho', \mathtt{e} : \rho)$ implies $fv(\mathtt{e}') \supseteq fv(\mathtt{e})$.)

Proof. Lemma 8.19 is shown by induction on the tree for the proof of evaluation or call in the pure λ-calculus and uses the observation about free variables. Proof is in the appendix. □

8.4. **The subexpression property.**

Definition 8.21. Given a state s in the extended language, we define its *expression support* $exp\_sup(s)$ by

$$exp\_sup(\mathtt{e} : \rho) = subexps(\mathtt{e}) \cup \bigcup_{\mathtt{x} \in fv(\mathtt{e})} exp\_sup(\rho(\mathtt{x}))$$

Lemma 8.22. (Subexpression property) *If $s \Downarrow s'$ or $s \to s'$ then $exp\_sup(s) \supseteq exp\_sup(s')$.*

Corollary 8.23. *If $\mathtt{P} : [] \Downarrow \lambda\mathtt{x.e} : \rho$ then $\lambda\mathtt{x.e} \in subexp(\mathtt{P})$. If $\mathtt{P} : [] \to^* \mathtt{e} : \rho$ then $\mathtt{e} \in subexps(\mathtt{P})$.*

The proof of Lemma 8.22 follows the same lines as the proof of Lemma 3.8. The proof for the rule (Gram) is immediate from the definition of subexpressions in the extended language. Proof omitted.

8.5. **Approximate extended semantics with size-change graphs.**

Definition 8.24. (Approximate evaluation and call rules for extended semantics with size-change graphs). The judgement forms are now $\mathtt{e} \to \mathtt{e}', G$ and $\mathtt{e} \Downarrow \mathtt{e}', G$, where $\mathtt{e}, \mathtt{e}' \in Exp_\Gamma$, and $source(G) = fv(e) \cup \{\epsilon\}$ and $target(G) = fv(e') \cup \{\epsilon\}$.

$$\frac{}{A \underset{n}{\to} \mathtt{e}, id_e^{=}} \; A ::= \mathtt{e} \in \Gamma \text{ (GramAG)} \qquad \frac{\mathtt{e} \underset{x}{\to} \mathtt{e}', G' \quad \mathtt{e}' \Downarrow v, G}{\mathtt{e} \Downarrow v, G'; G} \; x \in \{c,n\} \text{ (ResultAG)}$$

$$\frac{\mathtt{e}_1@\mathtt{e}_2 \in subexps(\mathtt{P}) \quad \mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0, G_1 \quad \mathtt{e}_2 \Downarrow v_2, G_2}{\mathtt{x} \Downarrow v_2, \{\mathtt{x} \xrightarrow{=} \epsilon\} \cup \{\mathtt{x} \xrightarrow{\downarrow} \mathtt{y} \mid \mathtt{y} \in fv(v_2)\}} \text{ (VarAG)}$$

$$\frac{}{\mathtt{e}_1@\mathtt{e}_2 \underset{r}{\to} \mathtt{e}_1, id^{\downarrow}_{\mathtt{e}_1}} \text{ (OperatorAG)} \qquad \frac{}{\mathtt{e}_1@\mathtt{e}_2 \underset{d}{\to} \mathtt{e}_2, id^{\downarrow}_{\mathtt{e}_2}} \text{ (OperandAG)}$$

$$\frac{}{\lambda\mathtt{x.e} \Downarrow \lambda\mathtt{x.e}, id^{=}_{\lambda\mathtt{x.e}}} \text{ (ValueAG)} \qquad \frac{\mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0, G_1 \quad \mathtt{e}_2 \Downarrow v_2, G_2}{\mathtt{e}_1@\mathtt{e}_2 \underset{c}{\to} \mathtt{e}_0, G_1^{-\epsilon/\lambda x.e_0} \cup_{e_0} G_2^{\epsilon \mapsto \mathtt{x}}} \text{ (CallAG)}$$

Putting the pieces together, we now show how to analyse any program in the regular grammar-extended λ-calculus . Let P be a program in the extended language.

Definition 8.25.
$absintExt(\mathtt{P}) =$

$$\{\; G_j \mid j > 0 \wedge \exists \mathtt{e}_i, G_i, (0 \le i \le j) : \mathtt{P} = \mathtt{e}_0 \wedge (\mathtt{e}_0 \to \mathtt{e}_1, G_1) \wedge \ldots \wedge (\mathtt{e}_{j-1} \to \mathtt{e}_j, G_j) \;\}$$

Theorem 8.26. *The set $absintExt(\mathtt{P})$ can be effectively computed from $\mathtt{P}$.*

Proof. In the extended λ-calculus there is only a fixed number of subexpressions of P, and a fixed number of of possible size-change graphs with

$$\text{source}, \text{target} \subseteq \{\epsilon\} \cup \{\mathtt{x} \mid \mathtt{x} \text{ is a variable that occurs in a subexpression of } \mathtt{P}\}$$

Thus $absintExt(\mathtt{P})$ can be computed in finite time by applying Definition 8.24 exhaustively, starting with P, until no new graphs or subexpressions are obtained. □

8.6. **Simulation properties of approximate extended semantics.** We will show the following properties of approximate extended semantics:

(1) Calls and evaluations for a program in *extended semantics with environments* can be stepwise simulated by *approximate extended semantics* with identical size-change graphs associated with corresponding calls and evaluations. To a call or evaluation in the extended λ-calculus with environments corresponds the same call or evaluation with environments removed.
(2) Suppose $\mathtt{P} \Rightarrow_\Gamma^* \mathtt{Q}$ for programs P,Q. Then calls and evaluations for Q in the pure lambda calculus with environments can be simulated by calls and evaluations in the approximate extended semantics for P using the relations $\Rightarrow_\Gamma^*$ and T.
(3) The extra edges in the size-change graphs in extended semantics can never give rise to incorrect termination analysis.

Lemma 8.27. *Let* P *be a program in the extended language and* $\mathtt{P} : [] \rightarrow^* \mathtt{e} : \rho$.
If $\mathtt{e} : \rho \rightarrow \mathtt{e}_0 : \rho_0, G$ *then* $\mathtt{e} \rightarrow \mathtt{e}_0, G$ *in approximate semantics.*
If $\mathtt{e} : \rho \Downarrow \mathtt{e}_0 : \rho_0, G$ *then* $\mathtt{e} \Downarrow \mathtt{e}_0, G$ *in approximate semantics.*

Proof. The proof is similar to the proof for approximation of the pure lambda-calculus 3.11 and 6.10. For rules (Value), (Operator), (Operand) it is immediate. The (Gram)-rule do not refer to the environment, hence the lemma holds if the (Gram)-rule has been applied. For rules (Call) and (Result) it holds by induction. For the (Var)-rule we need induction on the total size of the derivation, and we can argue as in the case of the pure lambda calculus. □

Lemma 8.28. *Let* P *be a program in the extended language and* Q *a program in the pure* λ*-calculus with* $\mathtt{P} \Rightarrow_\Gamma^* \mathtt{Q}$.

If $\mathtt{Q} : [] \rightarrow^* \mathtt{e} : \rho$ *and* $\mathtt{e} : \rho \Downarrow \mathtt{e}_0 : \rho_0, G$ *then there exist* $\mathtt{e}', \mathtt{e}_0', G'$ *with* $\mathtt{e}' \Rightarrow_\Gamma^* \mathtt{e}$, $\mathtt{e}_0' \Rightarrow_\Gamma^* \mathtt{e}_0$, $T(G', G)$ *such that* $\mathtt{P} \rightarrow^* \mathtt{e}'$ *and* $\mathtt{e}' \Downarrow \mathtt{e}_0', G'$.

If $\mathtt{Q} : [] \rightarrow^* \mathtt{e} : \rho$ *and* $\mathtt{e} : \rho \underset{x}{\rightarrow} \mathtt{e}_0 : \rho_0, G$, $x \in \{r, d, c\}$ *then there exist* $\mathtt{e}', \mathtt{e}_0', G'$ *with* $\mathtt{e}' \Rightarrow_\Gamma^* \mathtt{e}$,$\mathtt{e}_0' \Rightarrow_\Gamma^* \mathtt{e}_0$, $T(G', G)$ *such that* $\mathtt{P} \rightarrow^* \mathtt{e}'$ *and either* $\mathtt{e}' \underset{x}{\rightarrow} \mathtt{e}_0', G'$ *or* $\mathtt{e}' \underset{n}{\rightarrow} \mathtt{e}'' \underset{x}{\rightarrow} \mathtt{e}_0', G'$ *where in the last case* G' *is the composite size-change graph for the double call.*

Proof. The lemma follows from the simulation property lemma 8.19 together with lemma 8.27. □

Theorem 8.29.

(1) *Let* P *be a program in the extended language. If there is a program* Q *in the pure lambda-calculus such that* $\mathtt{P} \Rightarrow_\Gamma^* \mathtt{Q}$ *and there exists an infinite call-sequence in the call-graph for* Q *in the exact semantics, then there exists an infinite call-sequence with no infinitely descending thread in the call-graph for* P *in the approximate extended semantics.*
(2) *It follows that if each infinite call-sequence in the call-graphs for* P *in the approximate extended semantics has an infinitely descending thread, then* P *is* Γ*-terminating.*

Proof. (1): Assume an infinite call-sequence exists in the call-graph for Q. By the safety of the size-change graphs in the pure λ-calculus, the size-change graphs associated with this call sequence cannot have an infinitely descending thread. By lemma 8.28 there exists a simulating call-sequence in the call-graph for P such that the corresponding size-change graphs are in the T-relation. Let G_P, G_Q be any such two corresponding T-related size-change graphs from these call-sequences, $T(G_P, G_Q)$. By the definition of the T-relation

it holds that the largest subgraph of G_P, with *source* and *target* the same as $source(G_Q)$ and $target(G_Q)$, is equal to or a subset of G_Q. We need to show that the possible extra variables in the size-change graphs for the simulating sequence in the call-graph for `P` can never take part in an infinitely descending thread. By the definition of the T-relation it holds that an edge leaving from such a variable `x` must have have the form $(\texttt{x} \stackrel{=}{\rightarrow} \texttt{x})$ if any exists in the simulating sequence. Also by the definition of the T-relation, if $T(G_P, G_Q)$ and $(\texttt{x} \stackrel{=}{\rightarrow} \texttt{x}) \in G_P$ then $\texttt{x} \notin codomain(G_Q)$. Hence either an extra thread in the size-change graphs going out from `x` will be finite or it will be infinitely equal $\texttt{x} \stackrel{=}{\rightarrow} \texttt{x} \stackrel{=}{\rightarrow} \texttt{x} \stackrel{=}{\rightarrow} \ldots$, i.e. an extra variable can never take part in an infinitely descending thread in the simulating sequence.
(2) is a corollary to (1). □

Example 8.30. The following is an example of a program certified to terminate by our proof method. The program computes $x+2^n$ when applied to two arbitrary Church numerals for x and n. In Section 7 we analysed the program applied to Church numerals 3 and 4 (Example 7.2).

Grammar for Church numerals: C ::= λs.λz.A A ::= z | s@A
The program applied to two Church numerals:

```
[λn1.λn2.  n1                                    -- n --
           @   [λr.λa.11:(r@13:(r@a))]           -- g --
           @   [λ k.λ p.λ q.(p@((k@p)@q))]       - succ-
           @   n2 ]                              -- x --

@          C                                     -- Church numeral --
@          C                                     -- Church numeral --
```

Following is the output from program analysis. The analysis found the following loops from a program point to itself with the associated size-change graph and path. The first number refers to the program point, then comes a list of edges and last a list of numbers, the other program points that the loop passes through. The program points are found automatically by the analysis. The program points 30 and 32 are not written into the presentation of the program because they involve the subexpression A of a Church numeral. The subexpression associated with 30 is A and the subexpression associated with 32 is `s@A`. The loops from 30 to itself and from 32 to itself in the output correspond to the call sequence A→`s@A`→A→`s@A`....

```
SELF SCGS no repetition of graphs:

11 →* 11: [(r,>,r)]                                    []
11 →* 11: [(a,=,a),(r,>,r)]                            [13]
13 →* 13: [(a,=,a),(r,>,r)]                            [11]
13 →* 13: [(r,>,r)]                                    [11,11]
30 →* 30: [(ϵ,>,ϵ),(s,=,s),(z,=,z)]   [32]
32 →* 32: [(ϵ,>,ϵ),(s,=,s),(z,=,z)]   [30]

Size-Change Termination: Yes
```

9. Concluding matters

We have developed a method based on The Size-Change Principle to show termination of a closed expression in the untyped λ-calculus. This is further developed to analyse if a program in the λ-calculus will terminate when applied to any input from a given input set defined by a tree grammar. The analysis is safe and the method can be completely automated. We have a simple first implementation. The method certifies termination of many interesting recursive programs, including programs with mutual recursion and parameter exchange.

Acknowledgements. The authors gratefully acknowledge detailed and constructive comments by Arne Glenstrup, Chin Soon Lee and Damien Sereni, and insightful comments by Luke Ong, David Wahlstedt and Andreas Abel.

Appendix A. Proof of Lemma 2.6

Proof. $\Rightarrow$: Assume $\mathtt{P} \Downarrow$. To show: CT has no infinite call chain starting with $\mathtt{P}$. The proof is by induction on the height of the proof tree. Each call rule of 2.6 is associated with a use of rule (ApplyS) from Definition 2.2. So if $\mathtt{P}$ is a value, there is no call from $\mathtt{P}$. If $\mathtt{P} \Downarrow$ is concluded by rule (ApplyS), then $\mathtt{P} = \mathtt{e}_1@\mathtt{e}_2$ and by induction there is no infinite call chain starting with $\mathtt{e}_1$, $\mathtt{e}_2$ and $\mathtt{e}_0[v_2/\mathtt{x}]$. All call chains starting with $\mathtt{P}$ go directly to one of these. So, there are no infinite call chains starting with $\mathtt{P}$.

$\Leftarrow$: Assume CT has no infinite call chain starting with $\mathtt{P}$. To show: $\mathtt{P} \Downarrow$. Since the call tree is finitely branching, by König's lemma the whole call tree is finite, and hence there exists a finite number m bounding the length of all branches.

We prove that $\mathtt{e} \Downarrow$ for any expression in the call tree, by induction on the maximal length n of a call chain from $\mathtt{e}$.

$n = 0 : \mathtt{e}$ is an abstraction that evaluates to itself.

$n > 0 : \mathtt{e}$ must be an application $\mathtt{e} = \mathtt{e}_1@\mathtt{e}_2$. By rule (Operator) there is a call $\mathtt{e}_1@\mathtt{e}_2 \underset{d}{\rightarrow} \mathtt{e}_1$, and the maximal length of a call chain from $\mathtt{e}_1$ is less than n. By induction there exists v_1 such that $\mathtt{e}_1 \Downarrow v_1$. We now conclude by rule (Operand) that $\mathtt{e}_1@\mathtt{e}_2 \underset{r}{\rightarrow} \mathtt{e}_2$. By induction there exists v_2 such that $\mathtt{e}_2 \Downarrow v_2$.

All values are abstractions, so we can write $v_1 = \lambda\mathtt{x}.\mathtt{e}_0$. We now conclude by rule (Call) that $\mathtt{e}_1@\mathtt{e}_2 \underset{c}{\rightarrow} \mathtt{e}_0[v_2/\mathtt{x}]$. By induction again, $\mathtt{e}_0[v_2/\mathtt{x}] \Downarrow v$ for some v. This gives us all premises for the (ApplyS) rule of Definition 2.2, so $\mathtt{e} = \mathtt{e}_1@\mathtt{e}_2 \Downarrow v$. □

Appendix B. Proof of Lemma 3.11

Proof. To be shown:

$$\text{If } \mathtt{P} : [] \rightarrow^* \mathtt{e} : \rho \quad \text{and} \quad \mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho', \quad \text{then} \quad \mathtt{e} \Downarrow \mathtt{e}'.$$
$$\text{If } \mathtt{P} : [] \rightarrow^* \mathtt{e} : \rho \quad \text{and} \quad \mathtt{e} : \rho \rightarrow \mathtt{e}' : \rho', \quad \text{then} \quad \mathtt{e} \rightarrow \mathtt{e}'.$$

We prove both parts of Lemma 3.11 by course-of-value induction over the size $n = |\mathcal{D}|$ of a deduction $\mathcal{D}$ by Definition 3.3 of the assumption

$$\mathtt{P} : [] \rightarrow^* \mathtt{e} : \rho \wedge \mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho' \text{ or } \mathtt{P} : [] \rightarrow^* \mathtt{e} : \rho \wedge \mathtt{e} : \rho \rightarrow \mathtt{e}' : \rho'$$

The deduction size may be thought of as the number of steps in the computation of $\mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho'$ or $\mathtt{e} : \rho \rightarrow \mathtt{e}' : \rho'$ starting from $\mathtt{P} : []$.

The induction hypothesis $IH(n)$ is that the Lemma holds for all deductions of size not exceeding n. This implies that the Lemma holds for all calls and evaluations performed in the computation before the last conclusion giving ($\mathtt{P} : [] \rightarrow^* \mathtt{e} : \rho$ and $\mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho'$) or ($\mathtt{P} : [] \rightarrow^* \mathtt{e} : \rho$ and $\mathtt{e} : \rho \rightarrow \mathtt{e}' : \rho'$), i.e., the Lemma holds for premises of the rule last applied, and *for any call and evaluation in the computation until then.*

Proof is by cases on which rule is applied to conclude $\mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho'$ or $\mathtt{e} : \rho \rightarrow \mathtt{e}' : \rho'$. In all cases we show that some corresponding abstract interpretation rules can be applied to give the desired conclusion.

Base cases: Rule (Value), (Operator) and (Operand) in the exact semantics (def. 3.3) are modeled by axioms (ValueA), (OperatorA) and (OperandA) in the abstract semantics (def. 3.10). These are the same as their exact-evaluation counterparts, after removal of environments for (ValueA) and (OperatorA), and a premise as well for (OperandA). Hence the Lemma holds if one of these rules was the last one applied.

The (Var) rule is, however, rather different from the (VarA) rule. If (Var) was applied to a variable $\mathtt{x}$ then the assumption is ($\mathtt{P} : [] \rightarrow^* \mathtt{x} : \rho$ and $\mathtt{x} : \rho \Downarrow \mathtt{e}' : \rho'$). In this case $x \in dom(\rho)$ and $\mathtt{e}' : \rho' = \rho(x)$. The total size of the deduction (of both parts together) is n.

Now $\mathtt{P} : [] \rightarrow^* \mathtt{x} : \rho$ begins from the empty environment, and we know all calls are from state to state. The only possible way $\mathtt{x}$ can have been bound is by a previous use of the (Call) rule, the only rule that extends an environment.[5]

The premises of the (Call) rule require that operator and operand in an application have previously been evaluated. So it must be the case that there exist $\mathtt{e}_1@\mathtt{e}_2 : \rho''$ and $\lambda\mathtt{x}.\mathtt{e}_0 : \rho_0$ such that ($P : [] \rightarrow^* \mathtt{e}_1@\mathtt{e}_2 : \rho''$ and $\mathtt{e}_1 : \rho'' \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0$ and $\mathtt{e}_2 : \rho'' \Downarrow \mathtt{e}' : \rho'$) and the size of both deductions are strictly smaller than n. By the Subexpression Lemma, $\mathtt{e}_1@\mathtt{e}_2 \in subexp(\mathtt{P})$. By induction, Lemma 3.11 holds for both $\mathtt{e}_1 : \rho'' \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0$ and $\mathtt{e}_2 : \rho'' \Downarrow \mathtt{e}' : \rho'$, so $\mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0$ and $\mathtt{e}_2 \Downarrow \mathtt{e}'$ in the abstract semantics. Now we have all premises of rule (VarA), so we can conclude that $\mathtt{x} \Downarrow \mathtt{e}'$ as required.

For remaining rules (Apply) and (Call), when we assume that the Lemma holds for the premises in the rule applied to conclude $\mathtt{e} \Downarrow \mathtt{e}'$ or $\mathtt{e} \rightarrow \mathtt{e}'$, then this gives us the premises for the corresponding rule for abstract interpretation. From this we can conclude the desired result. □

Appendix C. Proof of Lemma 5.4

Proof. Define the length $L(\mathtt{e})$ of an expression e by:

$$L(\mathtt{x}) = 1 \quad L(\lambda\mathtt{x}.\mathtt{e}) = 1 + L(\mathtt{e}) \quad L(\mathtt{e}_1@\mathtt{e}_2) = 1 + L(\mathtt{e}_1) + L(\mathtt{e}_2)$$

For any expression $\mathtt{e}$, $L(\mathtt{e})$ is a natural number > 0. For a program, the length of the initial expression bounds all lengths of occurring expressions.

Define for a state s the height $H(s)$ of the state to be the height of the environment:

$$H(\mathtt{e} : \rho) = max\{(1 + H(\rho(\mathtt{x}))) \mid \mathtt{x} \in fv(\mathtt{e})\}$$

So, $H(\mathtt{e} : []) = 0$ the maximum of the empty set, and for any state $\mathtt{e} : \rho, H(\mathtt{e} : \rho)$ is a natural number ≥ 0. Let $>_{lex}$ stand for lexicographic order relation on pairs of natural

[5]This must have occurred in the part $\mathtt{P} : [] \rightarrow^* \mathtt{x} : \rho$.

numbers, hence $>_{lex}$ is well-founded. We prove that the relation $\succ$ on states is well-founded by proving that $\mathtt{e}_1 : \rho_1 \succ \mathtt{e}_2 : \rho_2$ implies that

$$(H(\mathtt{e}_1 : \rho_1), L(\mathtt{e}_1)) >_{lex} (H(\mathtt{e}_2 : \rho_2), L(\mathtt{e}_2))$$

First, consider $\succ_1$. Clearly, if $\mathtt{e}_1 : \rho_1 \succ_1 \mathtt{e}_2 : \rho_2$ then $H(\mathtt{e}_1 : \rho_1) > H(\mathtt{e}_2 : \rho_2)$. Hence even though $L(\mathtt{e}_2)$ might be larger than $L(\mathtt{e}_1)$, it holds that in the lexicographic order $(H(\mathtt{e}_1 : \rho_1), L(\mathtt{e}_1)) >_{lex} (H(\mathtt{e}_2 : \rho_2), L(\mathtt{e}_2))$.

Now, consider $\succ_2$. If $\mathtt{e}_1 : \rho_1 \succ_2 \mathtt{e}_2 : \rho_2$ then $H(\mathtt{e}_1 : \rho_1) \geq H(\mathtt{e}_2 : \rho_2)$ and $L(\mathtt{e}_1) > L(\mathtt{e}_2)$, hence in the lexicographic order $(H(\mathtt{e}_1 : \rho_1), L(\mathtt{e}_1)) >_{lex} (H(\mathtt{e}_2 : \rho_2), L(\mathtt{e}_2))$. Trivially, $\mathtt{e}_1 : \rho_1 = \mathtt{e}_2 : \rho_2$ implies $(H(\mathtt{e}_1 : \rho_1), L(\mathtt{e}_1)) =_{lex} (H(\mathtt{e}_2 : \rho_2), L(\mathtt{e}_2))$.

Recall, by definition $\succeq$ is the transitive closure of $\succ_1 \cup \succ_2 \cup =$, and $s_1 \succ s_2$ holds when $s_1 \succeq s_2$ and $s_1 \neq s_2$. So, from the derivations above we can conclude that $\mathtt{e}_1 : \rho_1 \succ \mathtt{e}_2 : \rho_2$ implies $(H(\mathtt{e}_1 : \rho_1), L(\mathtt{e}_1)) >_{lex} (H(\mathtt{e}_2 : \rho_2), L(\mathtt{e}_2))$, hence the relation $\succ$ on states is well-founded.

□

Appendix D. Proof of Theorem 6.8

Proof. For the "safety" theorem we use induction on proofs of $s \Downarrow s', G$ or $s \to s', G$. Safety of the constructed graphs for rules (ValueG), (OperatorG) and (OperandG) is immediate by Definitions 6.2 and 5.3.

In the following $\mathtt{x}, \mathtt{y}, \mathtt{z}$ are variables and p, q can be variables or ϵ.

The variable lookup rule **(VarG)** yields $\mathtt{x} : \rho \Downarrow \rho(\mathtt{x}), G$ with $G = \{\mathtt{x} \xrightarrow{\downarrow} \mathtt{y} \mid \mathtt{y} \in fv(\mathtt{e}')\} \cup \{\mathtt{x} \xrightarrow{=} \epsilon\}$ and $\rho(\mathtt{x}) = \mathtt{e}' : \rho'$. By Definition 5.2, $\overline{\mathtt{x} : \rho}(\mathtt{x}) = \overline{\rho(\mathtt{x})}(\epsilon)$, so arc $\mathtt{x} \xrightarrow{=} \epsilon$ satisfies Definition 6.2. Further, if $\mathtt{x} \xrightarrow{\downarrow} \mathtt{y} \in G$ then $\mathtt{y} \in fv(\mathtt{e}')$. Thus $\overline{x : \rho}(x) = \rho(\mathtt{x}) = e' : \rho' \succ \rho'(\mathtt{y}) = \overline{\rho(x)}(y)$ as required.

The rule **(CallG)** concludes $s \underset{c}{\to} s', G$, where $s = \mathtt{e}_1@\mathtt{e}_2 : \rho$ and $s' = \mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2]$ and $G = G_1^{-\epsilon/\lambda x.e_0} \cup_{e_0} G_2^{\epsilon \mapsto \mathtt{x}}$. Its premises are $\mathtt{e}_1 : \rho \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0, G_1$ and $\mathtt{e}_2 : \rho \Downarrow v_2, G_2$. We assume inductively that G_1 is safe for $(\mathtt{e}_1 : \rho, \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0)$ and that G_2 is safe for $(\mathtt{e}_2 : \rho, v_2)$. Let $v_2 = \mathtt{e}' : \rho'$.

We wish to show safety: that $p \xrightarrow{=} p' \in G$ implies $\overline{s}(p) = \overline{s'}(p')$, and $p \xrightarrow{\downarrow} p' \in G$ implies $\overline{s}(p) \succ \overline{s'}(p')$. By definition of $G_1^{-\epsilon/\lambda x.e_0}$ and $G_2^{\epsilon \mapsto \mathtt{x}}$, $p \xrightarrow{r} p' \in G = G_1^{-\epsilon/\lambda x.e_0} \cup_{e_0} G_2^{\epsilon \mapsto \mathtt{x}}$ breaks into 7 cases:

Case 1: $\mathtt{y} \xrightarrow{\downarrow} \mathtt{z} \in G_1^{-\epsilon/\lambda x.e_0}$ because $\mathtt{y} \xrightarrow{\downarrow} \mathtt{z} \in G_1$. By safety of G_1, $\overline{\mathtt{e}_1 : \rho}(\mathtt{y}) \succ \overline{\lambda\mathtt{x}.\mathtt{e}_0 : \rho_0}(\mathtt{z})$. Thus, as required,

$$\overline{s}(\mathtt{y}) = \overline{\mathtt{e}_1@\mathtt{e}_2 : \rho}(\mathtt{y}) = \overline{\mathtt{e}_1 : \rho}(\mathtt{y}) \succ \overline{\lambda\mathtt{x}.\mathtt{e}_0 : \rho_0}(\mathtt{z}) = \overline{\mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2]}(\mathtt{z}) = \overline{s'}(\mathtt{z})$$

Case 2: $\mathtt{y} \xrightarrow{=} \mathtt{z} \in G_1^{-\epsilon/\lambda x.e_0}$ because $\mathtt{y} \xrightarrow{=} \mathtt{z} \in G_1$. Like Case 1.

Case 3: $\mathtt{y} \xrightarrow{\downarrow} \epsilon \in G_1^{-\epsilon/\lambda x.e_0}$ because $\mathtt{y} \xrightarrow{r} \epsilon \in G_1$, then $x \notin fv(\mathtt{e}_0)$ by the definition of $G_1^{-\epsilon/\lambda x.e_0}$ and then $\mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2] = \mathtt{e}_0 : \rho_0$. By safety of G_1, $\overline{\mathtt{e}_1 : \rho}(\mathtt{y}) \succeq \overline{\lambda\mathtt{x}.\mathtt{e}_0 : \rho_0}(\epsilon) = \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0$. Thus, as required,

$$\overline{s}(\mathtt{y}) = \overline{\mathtt{e}_1@\mathtt{e}_2 : \rho}(\mathtt{y}) = \overline{\mathtt{e}_1 : \rho}(\mathtt{y}) \succeq \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0 \succ \mathtt{e}_0 : \rho_0 = \overline{s'}(\epsilon)$$

Case 4: $\epsilon \xrightarrow{\downarrow} p \in G_1^{-\epsilon/\lambda x.e_0}$ because $\epsilon \xrightarrow{r} p \in G_1$. Then it holds that either p is a variable-name or $\mathtt{x} \notin fv(\mathtt{e}_0)$. Now ϵ in G_1 refers to $\mathtt{e}_1 : \rho$, so $\mathtt{e}_1 : \rho \succeq \overline{\lambda \mathtt{x}.\mathtt{e}_0 : \rho_0}(p)$ by safety of G_1. Thus, as required,

$$\overline{s}(\epsilon) = \mathtt{e}_1@\mathtt{e}_2 : \rho \succ \mathtt{e}_1 : \rho \succeq \overline{\lambda \mathtt{x}.\mathtt{e}_0 : \rho_0}(p) \succeq \overline{\mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2]}(p) = \overline{s'}(p)$$

Case 5: $\mathtt{y} \xrightarrow{\downarrow} \mathtt{x} \in G$ because $\mathtt{x} \in fv(\mathtt{e}_0)$ and $\mathtt{y} \xrightarrow{\downarrow} \mathtt{x} \in G_2^{\epsilon \mapsto \mathtt{x}}$ because $\mathtt{y} \xrightarrow{\downarrow} \epsilon \in G_2$. By safety of G_2, $\overline{\mathtt{e}_2 : \rho}(\mathtt{y}) \succ \overline{v_2}(\epsilon)$. Thus, as required,

$$\overline{s}(\mathtt{y}) = \overline{\mathtt{e}_1@\mathtt{e}_2 : \rho}(\mathtt{y}) = \overline{\mathtt{e}_2 : \rho}(\mathtt{y}) \succ \overline{v_2}(\epsilon) = \overline{\mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2]}(\mathtt{x}) = \overline{s'}(\mathtt{x})$$

Case 6: $\mathtt{y} \xrightarrow{=} \mathtt{x} \in G$ because $x \in fv(e_0)$ and $\mathtt{y} \xrightarrow{=} \mathtt{x} \in G_2^{\epsilon \mapsto \mathtt{x}}$ because $\mathtt{y} \xrightarrow{=} \epsilon \in G_2$. Like Case 5.

Case 7: $\epsilon \xrightarrow{\downarrow} \mathtt{x} \in G$ because $\mathtt{x} \in fv(\mathtt{e}_0)$ and $\epsilon \xrightarrow{\downarrow} \mathtt{x} \in G_2^{\epsilon \mapsto \mathtt{x}}$ because $\epsilon \xrightarrow{r} \epsilon \in G_2$. By safety of G_2, $\overline{\mathtt{e}_2 : \rho}(\epsilon) = \mathtt{e}_2 : \rho$. Thus, as required,

$$\overline{s}(\epsilon) = \mathtt{e}_1@\mathtt{e}_2 : \rho \succ \mathtt{e}_2 : \rho \succeq \overline{v_2}(\epsilon) = \overline{\rho_0[\mathtt{x} \mapsto v_2]}(\mathtt{x}) = \overline{s'}(\mathtt{x})$$

The rule **(ApplyG)** concludes $s \Downarrow v, G'; G$ from premises $s \underset{c}{\rightarrow} s', G'$ and $s' \Downarrow v, G$, where $s = \mathtt{e}_1@\mathtt{e}_2 : \rho$ and $s' = \mathtt{e}' : \rho'$. We assume inductively that G' is safe for (s, s') and G is safe for (s', v). Let $G_0 = G'; G$.

We wish to show that G_0 is safe: that $p \xrightarrow{=} q \in G_0$ implies $\overline{s}(p) = \overline{v}(q)$, and $p \xrightarrow{\downarrow} q \in G_0$ implies $\overline{s}(p) \succ \overline{v}(q)$ (p, q can be variables or ϵ). First, consider the case $p \xrightarrow{=} q \in G_0$. Definition 4.2 implies $p \xrightarrow{=} p' \in G'$ and $p' \xrightarrow{=} q \in G$ for some p'. Thus by the inductive assumptions we have $\overline{s}(p) = \overline{s'}(p') = \overline{v}(q)$, as required.

Second, consider the case $p \xrightarrow{\downarrow} q \in G_0$. Definition 4.2 implies $p \xrightarrow{r_1} p' \in G'$ and $p' \xrightarrow{r_2} q \in G$ for some p', where either one or both of r_1, r_2 are $\downarrow$. By the inductive assumptions we have $\overline{s}(p) \succeq \overline{s'}(p')$ and $\overline{s'}(p') \succeq \overline{v}(q)$, and one or both of $\overline{s}(p) \succ \overline{s'}(p')$ and $\overline{s'}(p') \succ \overline{v}(q)$ hold. By Definition of $\succ$ and $\succeq$ this implies that $\overline{s}(p) \succ \overline{v}(q)$, as required.

□

Appendix E. Proof of Lemma 6.10

Proof. The rules are the same as in Section 3.10, only extended with size-change graphs. We need to add to Lemma 3.11 that the size-change graphs generated for calls and evaluations can also be generated by the abstract interpretation. The proof is by cases on which rule is applied to conclude $\mathtt{e} \Downarrow \mathtt{e}', G$ or $\mathtt{e} : \rho \rightarrow \mathtt{e}' : \rho', G$.

We build on Lemma 3.11, and we saw in the proof of this that in abstract interpretation we can always use a rule corresponding to the one used in exact computation to prove corresponding steps. The induction hypothesis is that the Lemma holds for the premises of the rule in exact semantics.

Base case (VarAG): By Lemma 3.11 we have $\mathtt{x} : \rho \Downarrow \mathtt{e}' : \rho'$ implies $\mathtt{x} \Downarrow \mathtt{e}'$. The size-change graph built in (VarAG) is derived in the same way from $\mathtt{x}$ and $\mathtt{e}'$ as in rule (VarG), and they will therefore be identical.

For other call- and evaluation rules without premises, the abstract evaluation rule is as the exact-evaluation rule, only with environments removed, and the generated size-change graphs are not influenced by environments. Hence the Lemma will hold if these rules are applied.

For all other rules in a computation: When we know that Lemma 3.11 holds and assume that Lemma 6.10 hold for the premises, then we can conclude that if this rule is applied, then Lemma 6.10 holds by the corresponding rule from abstract interpretation. □

Appendix F. Proof of Lemma 8.19

Proof. By induction on the tree for the proof of evaluation or call in the pure λ-calculus. Possible cases of the structure of $\mathtt{e}' : \rho'$ and $\mathtt{e} : \rho$ in S-related states:

$(\mathtt{x} : \rho',\ \mathtt{x} : \rho)$ $\quad$ $(\lambda\mathtt{x}.\mathtt{e}' : \rho',\ \lambda\mathtt{x}.\mathtt{e} : \rho)$ $\quad$ $(\mathtt{e}'_1@\mathtt{e}'_2 : \rho',\ \mathtt{e}_1@\mathtt{e}_2 : \rho)$
$(\mathtt{A} : \rho',\ \mathtt{x} : \rho)$ $\quad$ $(\mathtt{A} : \rho',\ \lambda\mathtt{x}.\mathtt{e} : \rho)$ $\quad$ $(\mathtt{A} : \rho',\ \mathtt{e}_1@\mathtt{e}_2 : \rho)$

Base cases, evaluations and calls in pure λ-calculus by rules without premisses.

Case $S(\mathtt{x} : \rho', \mathtt{x} : \rho)$: No calls from $\mathtt{x} : \rho$.

(Var)-rule, $\mathtt{x} : \rho \Downarrow \rho(\mathtt{x}) = \mathtt{e}_0 : \rho_0,\ \{\mathtt{x} \xrightarrow{=} \epsilon\} \cup \{\mathtt{x} \xrightarrow{\downarrow} \mathtt{y} \mid \mathtt{y} \in fv(\mathtt{e}_0)\}$ and $\mathtt{x} : \rho' \Downarrow \rho'(x) = \mathtt{e}'_0 : \rho'_0,\ \{\mathtt{x} \xrightarrow{=} \epsilon\} \cup \{\mathtt{x} \xrightarrow{\downarrow} \mathtt{y} \mid \mathtt{y} \in fv(\mathtt{e}'_0)\}$. Beginning from S-related states, by defintion of the relation S we have $S(\rho'(\mathtt{x}), \rho(\mathtt{x}))$ and $fv(\mathtt{e}'_0) \supseteq fv(\mathtt{e}_0)$. $source(G') = source(G)$ and the generation of size-change graphs gives that the restriction of G' to $target(G)$ equals G, hence $T(G', G)$.

Case $S(\lambda\mathtt{x}.\mathtt{e}' : \rho', \lambda\mathtt{x}.\mathtt{e} : \rho)$: No calls from $\lambda\mathtt{x}.\mathtt{e} : \rho$.
(Value)-rule, $\lambda\mathtt{x}.\mathtt{e} : \rho \Downarrow \lambda\mathtt{x}.\mathtt{e} : \rho, id^{=}_{\lambda\mathtt{x}.\mathtt{e}}$ and $\lambda\mathtt{x}.\mathtt{e}' : \rho' \Downarrow \lambda\mathtt{x}.\mathtt{e}' : \rho', id^{=}_{\lambda\mathtt{x}.\mathtt{e}'}$. $T(id^{=}_{\lambda\mathtt{x}.\mathtt{e}'}, id^{=}_{\lambda\mathtt{x}.\mathtt{e}})$.

Case $S(\mathtt{e}'_1@\mathtt{e}'_2 : \rho', \mathtt{e}_1@\mathtt{e}_2 : \rho)$:
(Operator)-rule, $\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{r}{\rightarrow} \mathtt{e}_1 : \rho, id^{\downarrow}_{e_1}$ and $\mathtt{e}'_1@\mathtt{e}'_2 : \rho' \underset{r}{\rightarrow} \mathtt{e}'_1 : \rho', id^{\downarrow}_{e'_1}$.Beginning from S-related states, by defintion of the relation S we have $S(\mathtt{e}'_1 : \rho', \mathtt{e}_1 : \rho)$. Then $T(id^{\downarrow}_{e'_1}, id^{\downarrow}_{e_1})$

Case $S(\mathtt{A} : \rho', \mathtt{x} : \rho)$:

(Var)-rule: $\mathtt{x} : \rho \Downarrow \rho(\mathtt{x}) = \mathtt{e}_0 : \rho_0, G$ where $G = \{\mathtt{x} \xrightarrow{=} \epsilon\} \cup \{\mathtt{x} \xrightarrow{\downarrow} \mathtt{y} \mid \mathtt{y} \in fv(\mathtt{e}_0)\}$. By the definition of S we must have $\mathtt{A} \Rightarrow^*_\Gamma x$. This againg by lemma 8.7 gives that we must have $\mathtt{A} ::= \mathtt{x}$. Then $\mathtt{A} : \rho' \underset{n}{\rightarrow} \mathtt{x} : \rho',\ id^{=}_x$ by (Gram)-rule, and we have $S(\mathtt{x} : \rho', \mathtt{x} : \rho)$. Also $\mathtt{x} : \rho' \Downarrow \rho'(\mathtt{x}) = \mathtt{e}'_0 : \rho'_0, G''$ where $G'' = \{\mathtt{x} \xrightarrow{=} \epsilon\} \cup \{\mathtt{x} \xrightarrow{\downarrow} \mathtt{y} \mid \mathtt{y} \in fv(\mathtt{e}'_0)\}$ by (Var)-rule. The edges in G'' are the same as the edges in $G' = id^{=}_x; G''$. Hence by (Result)-rule $A \Downarrow \rho'(\mathtt{x}), G'$. As before $S(\rho'(\mathtt{x}), \rho(\mathtt{x}))$ and $T(G', G)$.

Cases $S(\mathtt{A} : \rho', \lambda\mathtt{x}.\mathtt{e} : \rho)$ with (Value)-rule, and $S(\mathtt{A} : \rho', \mathtt{e}_1@\mathtt{e}_2 : \rho)$ with (Operator)-rule: Similarly by use of lemma 8.7 and reasoning as above. We will use the rules (Gram)(Value) (Result) and (Gram)(Operator) respectively, where (Value) and (Operator) do not have premises.

Step cases.

Case $S(\mathtt{e}'_1@\mathtt{e}'_2 : \rho', \mathtt{e}_1@\mathtt{e}_2 : \rho)$. $\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{d}{\rightarrow} \mathtt{e}_2 : \rho, id^{\downarrow}_{\mathtt{e}_2}$ by (Operand)-rule. It follows from the definition of S that also $S(\mathtt{e}'_1 : \rho', \mathtt{e}_1 : \rho)$ hence by IH since $\mathtt{e}_1 : \rho \Downarrow$ then also $\mathtt{e}'_1 : \rho' \Downarrow$ and then $\mathtt{e}'_1@\mathtt{e}'_2 : \rho' \underset{d}{\rightarrow} \mathtt{e}'_2 : \rho', id^{\downarrow}_{\mathtt{e}'_2}$ and by the definition of S we have $S(\mathtt{e}'_2 : \rho', \mathtt{e}_2 : \rho)$,

$T(id^{\downarrow}_{\mathtt{e}'_2}, id^{\downarrow}_{\mathtt{e}_2})$.

The next case is the one that requires the most consideration to see that we stay within the T-relation. Assume we know for graphs $\tilde{G}', \tilde{G}$, that the restriction of $\tilde{G}'$ to source and target of $\tilde{G}$ is a subset of $\tilde{G}$. Notice, if $x, y \in source(\tilde{G}') \setminus source(\tilde{G})$ and $\mathtt{x}, \mathtt{z} \in target(\tilde{G}') \setminus target(\tilde{G})$, then for testing $T(\tilde{G}', \tilde{G})$ we only need to look at which edges leaves from $\mathtt{x}, \mathtt{y}$, we do not need to care about if other edges goes into $\mathtt{x}, \mathtt{z}$.

Case $S(\mathtt{e}'_1@\mathtt{e}'_2 : \rho', \mathtt{e}_1@\mathtt{e}_2 : \rho)$. $\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{c}{\rightarrow} \mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2], G_1^{-\epsilon/\lambda\mathtt{x}.\mathtt{e}_0} \cup_{\mathtt{e}_0} G_2^{\epsilon\mapsto\mathtt{x}}$ by (Call)-rule, where we have the premises $\mathtt{e}_1 : \rho \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0, G_1$ and $\mathtt{e}_2 : \rho \Downarrow v_2, G_2$.

It follows from the definition of S that also $S(\mathtt{e}'_1 : \rho', \mathtt{e}_1 : \rho)$ and $S(\mathtt{e}'_2 : \rho', \mathtt{e}_2 : \rho)$. Hence by IH since $\mathtt{e}_1 : \rho \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0, G_1$ then also $\mathtt{e}'_1 : \rho' \Downarrow v, G'_1$ where $T(G'_1, G_1)$ and $S(v, \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0)$. Then by definition of values, relations $\Rightarrow^*_\Gamma$ and S we must have $v = \lambda\mathtt{x}.\mathtt{e}'_0 : \rho'_0$. Also by IH since $\mathtt{e}_2 : \rho \Downarrow v_2, G_2$ then also $\mathtt{e}'_2 : \rho' \Downarrow v'_2, G'_2$ where $T(G'_2, G_2)$ and $S(v'_2, v_2)$. Then we have the premises to conclude $\mathtt{e}'_1@\mathtt{e}'_2 : \rho' \underset{c}{\rightarrow} \mathtt{e}'_0 : \rho'_0[\mathtt{x} \mapsto v'_2], G_1'^{-\epsilon/\lambda x.e'_0} \cup_{e'_0} G_2'^{\epsilon\mapsto\mathtt{x}}$. By definition of S we have $S(\mathtt{e}'_0 : \rho'_0[\mathtt{x} \mapsto v'_2], \mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2])$. We notice that $\mathtt{x} \notin fv(\lambda\mathtt{x}.\mathtt{e}'_0)$ and therefore $(p \xrightarrow{r} \mathtt{x}) \notin G'_1$.
We consider different possibilities for the generated graphs:

If $\mathtt{x} \in fv(\mathtt{e}'_0)$ but $\mathtt{x} \notin fv(\mathtt{e}_0)$ then we can have some extra edges going to $\mathtt{x}$ in extended semantics where we will have no edges to $\mathtt{x}$ in pure semantics because $\mathtt{x}$ is not in the target, but this is acceptable in the T-relation. There can also be some extra edges going to ϵ in pure semantics where no edges go to ϵ in exact semantics, but as ϵ is within the codomain in pure semantics, this is also acceptable in the T-relation. Since $T(G'_1, G_1)$ it will still hold that $T(G_1'^{-\epsilon/\lambda x.e'_0} \cup_{e'_0} G_2'^{\epsilon\mapsto\mathtt{x}}, G_1^{-\epsilon/\lambda x.e_0} \cup_{e_0} G_2^{\epsilon\mapsto\mathtt{x}})$.

If $\mathtt{x} \in fv(\mathtt{e}_0)$ then also $\mathtt{x} \in fv(\mathtt{e}'_0)$ and if $\mathtt{x} \notin fv(\mathtt{e}'_0)$ then $\mathtt{x} \notin fv(\mathtt{e}_0)$, in these cases since $T(G'_1, G_1)$ and $T(G'_2, G_2)$ also $T(G_1'^{-\epsilon/\lambda x.e'_0} \cup_{e'_0} G_2'^{\epsilon\mapsto\mathtt{x}}, G_1^{-\epsilon/\lambda x.e_0} \cup_{e_0} G_2^{\epsilon\mapsto\mathtt{x}})$.

Case $S(\mathtt{A} : \rho', \mathtt{e}_1@\mathtt{e}_2 : \rho)$ with $\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{d}{\rightarrow} \mathtt{e}_2 : \rho, id^{\downarrow}_{\mathtt{e}_2}$ by(Operand)-rule. By the definition of S we must have $\mathtt{A} \Rightarrow^*_\Gamma \mathtt{e}_1@\mathtt{e}_2$. This againg by lemma 8.7 gives that we must have $\mathtt{A} ::= \mathtt{e}'_1@\mathtt{e}'_2$. Then $\mathtt{A} : \rho' \underset{n}{\rightarrow} \mathtt{e}'_1@\mathtt{e}'_2 : \rho',\ id^{=}_{e'_1@e'_2}$ by (Gram)-rule, and we have $S(\mathtt{e}'_1@\mathtt{e}'_2 : \rho', \mathtt{e}_1@\mathtt{e}_2 : \rho)$. Then we have seen that $\mathtt{e}'_1@\mathtt{e}'_2 : \rho' \underset{d}{\rightarrow} \mathtt{e}'_2 : \rho', id^{\downarrow}_{\mathtt{e}'_2}$ with $S(\mathtt{e}'_2 : \rho', \mathtt{e}_2 : \rho)$, $T(id^{\downarrow}_{\mathtt{e}'_2}, id^{\downarrow}_{\mathtt{e}_2})$ and we have that the edges of $id^{\downarrow}_{\mathtt{e}'_2}$ are the same as the edges of $(id^{=}_{e'_1@e'_2}; id^{\downarrow}_{\mathtt{e}'_2})$ hence $T((id^{=}_{e'_1@e'_2}; id^{\downarrow}_{\mathtt{e}'_2}), id^{\downarrow}_{\mathtt{e}_2})$.

Case $S(A : \rho', e_1@e_2 : \rho)$ with (Call)-rule $e_1@e_2 : \rho \underset{c}{\rightarrow} e_0 : \rho_0[x \mapsto v_2], G$: Similarly as before we have $A : \rho' \underset{n}{\rightarrow} e'_1@e'_2 : \rho',\ id^{=}_{e'_1@e'_2}$ by (Gram)-rule, and we have $S(e_1@e_2 : \rho, e'_1@e'_2 : \rho')$. We can now use the derivation above and with the notation from above we have $e'_1@e'_2 : \rho' \underset{c}{\rightarrow} e'_0 : \rho'_0[x \mapsto v'_2], G'$ with $S(e'_0 : \rho'_0[x \mapsto v'_2], e_0 : \rho_0[x \mapsto v_2])$ and $T(G', G)$. Looking into the derivation of G' we find that the edges of G' are the same as the edges of $(id^{=}_{e'_1@e'_2}; G')$.

Case $S(\mathtt{e}' : \rho', \mathtt{e} : \rho)$, $\mathtt{e} : \rho \Downarrow v, G$ by (Result)-rule, where we have the premises $\mathtt{e} : \rho \underset{c}{\rightarrow} \mathtt{e}_s : \rho_s, G_s$ and $\mathtt{e}_s : \rho_s \Downarrow v, G_v$, $G = G_s; G_v$: By IH since $e : \rho \underset{c}{\rightarrow} \mathtt{e}_s : \rho_s, G_s$ then

$\mathsf{e}' : \rho' \underset{n}{\rightarrow}{}^{j} s : \rho' \underset{c}{\rightarrow} \mathsf{e}'_s : \rho'_s, G'_s$ with $S(\mathsf{e}'_s : \rho'_s, \mathsf{e}_s : \rho_s)$, and $T(G'_s, G_s)$, $j \in \{0, 1\}$. Again by IH since $\mathsf{e}_s : \rho_s \Downarrow v, G_v$ then $\mathsf{e}'_s : \rho'_s \Downarrow v', G'_v$ with $S(v, v')$ and $T(G'_v, G_v)$. Let $G' = G'_s; G'_v$ then $T(G', G)$. If $j = 0$ we have the premises to conclude $\mathsf{e}' : \rho' \Downarrow v', G'$. If $j = 1$ by lemma 8.17 we have $S(s : \rho', \mathsf{e} : \rho)$ and we have the premises to conclude $s : \rho' \Downarrow v', G$, and by applications of (Result)-rule once more in the extended semantics we can also conclude $\mathsf{e}' : \rho' \Downarrow v', id_s^{=}; G'$ where the edge set of G' is the same as the edge set of $id_s^{=}; G'$. □

References

[1] Abel, A.: 2004, Termination Checking with Types. *RAIRO - Theoretical Informatics and Applications, Special Issue: Fixed Points in Computer Science (FICS'03)* 38(4), 277–319.

[2] Abel, A.: 2006 A Polymorphic Lambda-Calculus with Sized Higher-Order Types. Ph.d. thesis, Ludwig-Maximilians-Universität München

[3] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, T. Uustalu. Type-based termination of recursive definitions. Mathematical. Structures in Comp. Sci. 14:97–141, 2004.

[4] Cousot, P. and R. Cousot: 1977, Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In: *POPL 77: 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 238–252.

[5] C.C. Frederiksen and N.D. Jones. Running-time Analysis and Implicit Complexity. unpublished, 2006.

[6] T. Arts and J. Giesl. Termination of Term Rewriting Using Dependency Pairs. Theoretical Computer Science 236:133–178, 2000.

[7] Giesl, J., R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. Technical report, RWTH Aachen, 2005.

[8] J. Giesl, S. Swiderski, P. Schneider-Kamp, R. Thiemann. Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages. In RTA 2006: Rewriting Techniques and Applications: (Frank Pfenning, eds), pp. 297-312. Volume 4098 of Lecture Notes in Computer Science, 2006.

[9] J.Y. Girard, Y. Lafont, P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[10] N.D. Jones and A. Glenstrup. Partial Evaluation Termination Analysis and Specialization-Point Insertion. ACM Transactions on Programming Languages and Systems: 27, 6: 1147–1215, 2005.

[11] N.D. Jones: Flow Analysis of Lambda Expressions, *ICALP 1981*, *Lecture Notes in Computer Science*. Springer-Verlag (1981).

[12] N.D. Jones and N. Bohr. Termination analysis of the untyped λ-calculus. In RTA 2004: Rewriting Techniques and Applications: (V. van Oostrom, eds.), pp. 1–23. Volume 3091 of Lecture Notes in Computer Science. Springer. June, 2004.

[13] N.D. Jones and F. Nielson. Abstract Interpretation: a Semantics-Based Tool for Program Analysis. In Handbook of Logic in Computer Science, pp. 527-629. Oxford University Press, 1994.

[14] C.S. Lee, N.D. Jones and A.M. Ben-Amram The Size-Change Principle for Program Termination POPL 2001: Proceedings 28^{th} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 2001.

[15] C.S. Lee. Finiteness analysis in polynomial time. In Static Analysis: 9th International Symposium, SAS 2002 (M Hermenegildo and G Puebla, eds.), pp. 493-508. Volume 2477 of Lecture Notes in Computer Science. Springer. September, 2002.

[16] C.S. Lee. Program termination analysis in polynomial time. In Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002 (D Batory, C Consel, and W Taha, eds.), pp. 218-235. Volume 2487 of Lecture Notes in Computer Science.

[17] C.S. Lee. Program Termination Analysis and the Termination of Offline Partial Evaluation Ph.D. thesis, University of Western Australia, March 2001.

[18] G.D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. Theoretical Computer Science, 1, 1975.

[19] D. Sereni, N.D. Jones. Termination Analysis of Higher-Order Functional Programs In APLAS 2005: The Third Asian Symposium on Programming Languages and Systems (Kwangkeun Yi, ed.), pp. 281–297. Volume 3780 of Lecture Notes in Computer Science. Springer. November, 2005.
[20] D. Sereni. λ-SCT and simple types. E-mail communication. April, 2005.
[21] D. Sereni. Termination Analysis of Higher-Order Functional Programs D.Phil thesis, OUCL (Oxford University Computing Laboratory), 2006.
[22] D. Olin Shivers. Control-Flow Analysis of Higher-Order Languages D.Phil thesis, Carnegie Mellon University, 1991.
[23] D. Olin Shivers. Higher-order control-flow analysis in retrospect: Lessons learned, lessons abandoned In *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection*, pages 269–270, editor Kathryn S. McKinley, 2004.
[24] W.W. Tait. Intensional interpretation of functionals of finite type I. Journal of Symbolic Logic 32:198–212, 1967.
[25] Toyama, Y., Termination of S-expression rewriting systems: Lexicographic path ordering for higher-order terms. In: *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA 2004)*, Vol. 3091 of *Lecture Notes in Computer Science*. pp. 40–54, 2004.
[26] D. Wahlstedt. Type Theory with First-Order Data Types and Size-Change Termination. Licentiate thesis, Chalmers University of Technology, Gothenburg, Sweden, 2004.
[27] D. Wahlstedt. Dependent Type Theory with Paraetrized First-Order Data Types and Well-Founded Recursion. Ph.D. thesis, Chalmers University of Technology, Gothenburg, Sweden, 2007.
[28] Hongwei Xi. Dependent Types for Program Termination Verification, Journal of Higher-Order Symbolic Logic, 15(1), pp. 91–131, 2002.